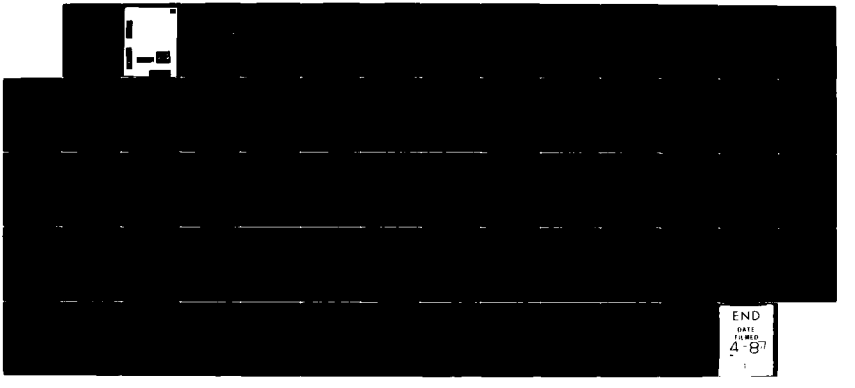
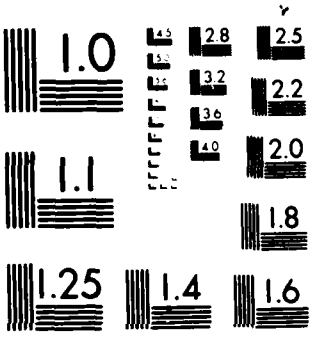
AD-A178 404 A SIMULATION ENVIRONMENT FOR SCHEMA(U) MASSACHUSETTS 1/1
INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE
M A ST. PIERRE DEC 86 MIT/LCS/TR-386 N00014-80-C-0622
UNCLASSIFIED F/G 9/3 NL



END
DATE FILMED
4-87



(12)

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-386

A SIMULATION ENVIRONMENT FOR SCHEMA

Margaret Ann St. Pierre

December 1986

545 TECHNOLOGY SQUARE CAMBRIDGE, MASSACHUSETTS 02139

87 3

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

A178404 12

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release;distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-386	5. MONITORING ORGANIZATION REPORT NUMBER(S) DARPA/DOD N00014-80-C-0622
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO

DTIC
MAR 2 3 1987
E

11. TITLE (Include Security Classification) A Simulation Environment for Schema			
12. PERSONAL AUTHOR(S) St. Pierre, Margaret Ann			
13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1986 December	15 PAGE COUNT 63
16. SUPPLEMENTARY NOTATION			

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	CAD, VLSI, simulation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

In present day circuit design, many independent simulation tools are available for analyzing circuits at various levels of detail. This thesis presents a framework to tie these tools into the Simulation Environment in Schema, an integrated CAD system. The framework easily incorporates additional simulators, serves as a foundation upon which to build new analysis tools, and provides the ability for mixed-mode simulation. The Simulation Environment is composed of common data representations, a Generic Simulator, and a single user interface. A common representation for topological, model, and waveform data objects facilitates a uniform interface to the user and to all CAD tools. The Generic Simulator coordinates the flow of data objects between each simulator and the user or analysis tool.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION
22a NAME OF RESPONSIBLE INDIVIDUAL Judy Little	22b TELEPHONE (Include Area Code) (617)523-5894 22c OFFICE SYMBOL

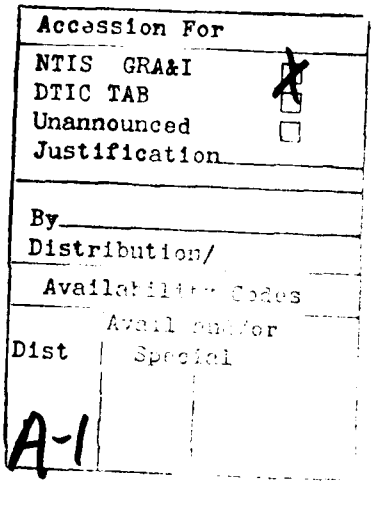
DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

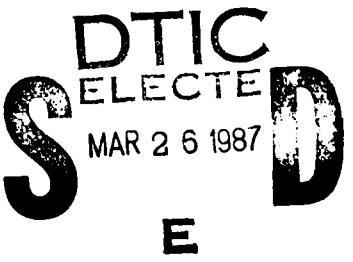
A Simulation Environment for Schema

by

Margaret Ann St. Pierre



Support for this research was provided by the Defense Advanced Research Projects Agency of the Department of Defense under Contract No. N00014-80-C-0622.



A Simulation Environment for Schema

by

Margaret Ann St. Pierre

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for
the degrees of Master of Science and Electrical Engineer

Abstract

In present day circuit design, many independent simulation tools are available for analyzing circuits at various levels of detail. This thesis presents a framework to tie these tools into the Simulation Environment in Schema, an integrated CAD system. The framework easily incorporates additional simulators, serves as a foundation upon which to build new analysis tools, and provides the ability for mixed-mode simulation. The Simulation Environment is composed of common data representations, a Generic Simulator, and a single user interface. A common representation for topological, model, and waveform data objects facilitates a uniform interface to the user and to all CAD tools. The Generic Simulator coordinates the flow of data objects between each simulator and the user or analysis tool.

Thesis Supervisor: Professor Richard Zippel
Title: Associate Professor of Electrical Engineering and Computer Science

Key Words and Phrases: CAD, VLSI, simulation

Acknowledgments

I would like to thank:

My thesis advisor, Rich Zippel, for the inspiration, focus, and encouragement that made this thesis possible, for fathering the famous ski resorts upon which this thesis work was implemented, and for providing me with quiet officespace.

Brian Williams for many stimulating discussions and suggestions along the way.

Jeff Arnold, Randy Davis, Steve Heller, and Jerry Roylance for comments on early drafts of this thesis.

George Clark and Mike MacDonald for giving unity to Schema.

My friends back at the Schema Chalet.

Moses Ma, Peter Nuth, and Pete Osler for interesting non-technical discussions.

Jim Restivo for faithfully escorting me to and beyond the finish line.

My family for their love and support throughout my many years of academia.

Table of Contents

Table of Figures

Chapter One

Introduction

Circuit design requires the assistance of a comprehensive range of computer aided design (CAD) tools, many of which either currently exist or are under development. Individually, each tool addresses a specific task in the design process. As an *integrated* collection, however, the tools *share data and tasks* across all stages of the design process. Unfortunately, no one system has effectively integrated the collection into a single design environment.

Research on such an environment is presently underway at M.I.T. with the development of Schema [Ziopel 85]. Schema research focuses on providing a software environment for easily integrating all CAD tools necessary for design and allowing the effortless building of new tools into the existing system. One area of major interest in Schema and of circuit design in general is simulation. Fatal design errors are detected and circuit performance is measured by simulating the operation of electronic designs. In this way, simulation invaluably contributes to the success of high-performance circuit designs and is a vital component of any CAD system.

This thesis presents a **Simulation Environment for Schema** following in the footsteps of the integrated software design environment established in Schema.

1.1 Motivation

Many simulators have been developed to satisfy different design needs using a single modeling level of circuit abstraction. Often the designer is overwhelmed by the need to learn the operation of and to manually recode circuit descriptions for each individual simulator. In addition, output waveforms associated with one particular circuit module's simulation must be interpreted and manually translated for use as input to some other interconnected module's simulation. Because of the massive time investment required, this process is typically omitted altogether.

The recent trend has been towards mixed-mode simulation whereby different levels of simulation are consolidated into one software package. At the high end, the Sable [Hill 79, Hill 80]

system combines behavioral, register transfer, and gate level descriptions. Similarly, Themis [Doshi 84] addresses simulation at the behavioral, register transfer, logic, and switch levels. Both simulators deal exclusively in the digital domain, however; neither includes circuit, timing, or linear level models, which are critical to the design of high-performance circuits. On the low end, concurrent circuit, timing, and logic analyses are illustrated in both the Diana [Arnout 78, Antognetti 84] and Splice [Newton 78, Newton 79] systems. In addition, the second generation Motis [Chawla 75, Fan 77, Chen 84, Antognetti 84] program combines timing, switch, and logic level simulators into one software package, running on a single mainframe; accuracy is reduced by omitting the detailed transistor models available in a circuit level simulator such as Spice2 [Nagel 75, Cohen 76].

These and others [Nestor 82, Thomas 83, Borrione 83, Lanthrop 85] are attempts to combine simulators using a range of modeling levels into a single software program. One disadvantage of this single system approach is a loss in *computational efficiency*. With increasing integrated circuit complexity, the computational power required for simulating very large circuits becomes a major bottleneck to the design effort. Even the use of the most advanced hardware and software technology inevitably results in extensive execution times for a single system. Expensive design effort is halted while waiting upon simulation results. Another cost is incurred from discarding old, yet still usable simulators to invest in software recoding for a mixed-mode system. For example, in an effort to provide an integrated computer aided design system for Sandia, a substantial amount of manpower was invested in understanding, recoding, and debugging undocumented industry and university software programs [Daniel 82].

With the accelerated advancement of today's technology, new simulators are continually being developed using state-of-the-art hardware technology, and more efficient, optimized, and sophisticated algorithms. Dramatic speed improvements are achievable with special-purpose hardware, such as the Yorktown Engine [Pfister 82] and the Logic Simulation Machine [Abramovici 83], and highly parallel algorithms, such as Prsim [Arnold 85] and Msplice [Deutsch 84] designed specifically for multi-processor systems.

Simulation alone cannot guarantee the success of today's high performance circuits. In conjunction with simulation, *analysis tools* are an essential ingredient of the design process. Analysis tools operate on simulation data. This data may pertain to one simulation, multiple simulations, or ultimately different simulation levels. Performance evaluation, verification of

simulation results against specifications, and circuit partitioning for different levels of simulation are just a sampling of invaluable analysis tools. Analysis tools also instigate simulations. An analysis tool may schedule a series of simulations to compare the performance of different designs or the behavior of a single design in different operating regimes.

Each analysis tool is simple to build, yet creating a simulator and user interface for each is a major undertaking. In effect, the existence of the analysis tool alone is not justified. For example, mathematical operations on waveforms are useful for analyzing circuit simulation results. For instance, power consumption over time amounts to a simple multiplication of waveforms, yet without a graphical user interface and a simulator interface, the tool is unusable. The designer would be forced to manually enter the simulation data points - a tedious, error-prone, and time-consuming task - as well as interpret the numerical output data.

1.2 Design Goals

A framework is essential to tie simulation tools into a common environment. This thesis presents such a framework: ***A Simulation Environment for Schema***. The framework is designed to easily integrate simulation tools, to serve as a foundation for building new analysis tools, and to provide mixed-mode capability. The following sections detail each of these design goals.

1.2.1 Integrating Simulation Tools

The Simulation Environment is designed with the ability to readily integrate new as well as currently existing simulation tools. Simulation of all modeling levels may be easily incorporated; this includes tools exploiting each of the various transistor modeling levels and the simulators that address the more abstract circuit representations. Without slowing down the user's design effort, simulation can be *distributed* to another local process or remote engine that can efficiently run the simulation. Distributing the effort among whatever engines are currently available, and potentially least loaded, enhances the overall computational power of the designer's environment. Furthermore, because a large amount of time, money, and effort went into developing and maintaining the existing simulation tools, they could remain constantly in use - greatly enhancing computational power. Adding new simulators allows the environment to keep pace with the rapid development of new hardware and software simulation engines.

1.2.2 Building Analysis Tools

The Simulation Environment could serve as a foundation for building an unlimited number of powerful analysis tools. Automatic partitioning algorithms can be developed for partitioning large-scale circuits into a collection of blocks to be individually simulated at different modeling levels. Another tool could schedule a series of simulations for each block to verify that it meets specifications. Additionally, small analysis tools could be designed to compare the results of different simulations or to perform operations on simulation output. Comparison and evaluation of the performance of new simulators could even be executed by an analysis tool.

1.2.3 Mixed-mode Capability

Mixed-mode refers to transforming the output data from one module's simulation for use as input to an interconnected module's simulation, where each module may be modeled at different levels of detail. For example, certain portions of a design may require the accuracy of a circuit simulation, while for other less critical portions, a less exact switch or logic level simulation is most appropriate. Both require simulation, yet using different simulators. With the mixed-mode capability, the Simulation Environment transforms the output analog waveforms from the circuit simulation into logic waveforms for use in the switch or logic level simulation, and vice versa.

1.3 Overview of Thesis

Chapter 2 opens with a brief overview of the types of simulators available. This naturally leads into a discussion of the goals of the Simulation Environment and the approach taken to achieve them. Next each component of the Simulation Environment is briefly described: the uniform data representations, a Generic Simulator, and a common user interface. The chapter closes with a discussion of the techniques available in Schema that are useful to the Simulation Environment.

The circuit topology, models, and waveforms are the data required for simulation. Their uniform representations and user interface are discussed in Chapters 3, 4, and 5, respectively. When integrating additional simulators or building new analysis tools, only new data types and local operations need to be defined as described in the latter sections of each chapter.

Chapter 6 describes the role of the Generic Simulator in the Simulation Environment. The

Generic Simulator contains the simulation tools of the environment and *generically* interfaces them to the objects in the environment, to the user, and to the analysis tools. This chapter presents each step of the Generic Simulation Process.

Chapter 7 concludes with a summary of the Simulation Environment for Schema recounting the properties achieved. Suggestions for possible future analysis tools are cited. These tools could be easily built on top of the Simulation Environment in Schema.

Chapter Two

Design Methodology

The currently available simulators are reviewed with respect to input and output data required for each. Next, a design strategy is developed to tie these simulators into a single Simulation Environment. Each component of the Simulation Environment is defined along with its corresponding role in the simulation process. And finally, the implementation of the Simulation Environment within Schema is presented.

2.1 Simulation Domain

Many simulators have been developed to satisfy different design needs throughout the various stages of the design process. This section briefly describes the different kinds of simulators in use today. Notably, each simulator utilizes different algorithms, accepts input such as a circuit description, excitation signals and perhaps some modeling parameters, and ultimately produces output data.

Circuit simulators provide the most detailed level of simulation; node voltages and branch current waveforms are calculated and plotted. *General purpose* circuit simulators, such as Spice2 [Nagel 75, Cohen 76] and Astap [Weeks 73], apply general algorithms for non-linear static, linear ac, and non-linear transient analyses. Circuits may contain capacitors, resistors, inductors, mutual inductors, voltage and current sources, and a wide range of nonlinear active devices including diodes, bipolar junction transistors (BJTs), junction field-effect transistors (JFETs), and metal-oxide-semiconductor (MOS) field-effect transistors (FETs). Each semiconductor device is modeled with a set of process parameters. Spice2, for example, has three built-in types of MOS device models: Shichman and Hodges, analytical, and semi-empirical models. At this level of detail, circuit simulators are generally cost effective for circuits with a few hundred devices or less. Execution time can be increased by replacing analytic device models with simplified table look-up models relating device current to terminal voltages. These general-purpose circuit simulators are largely independent of technology. If simulation algorithms are tailored to specific

technologies or applications, substantial speed improvements can be achieved. To take advantage of the unilateral nature of MOS devices, *relaxation-based* circuit simulation [Dumlugol 83, Newton 84] algorithms provide up to a twofold increase in simulation speed over general-purpose circuit simulators.

The *linear-model simulator* Rsim [Terman 83] represents MOS transistors as resistors in series with a voltage-controlled switch. This model provides logical and approximate timing information. Logic behavior is determined by a fast event-driven algorithm; transition times depend upon on effective transistor resistance, and interconnect and gate capacitance. Using this simplified linear model, networks containing up to 50,000 transistors may be simulated. Instead of node voltages and branch currents, discrete logic states at network nodes are used.

Switch-level simulators such as Mossim [Bryant 81] and Esim [Terman 83] model MOS transistors as a network of on/off switches. This model captures the logical properties of a circuit while ignoring many of the detailed electrical issues. A switching network is most appropriate for simulating the bidirectional nature of MOS transistors. Furthermore, since so little modeling information is retained for each transistor, this type of simulator is able to handle larger scale designs. Signals are typically represented in terms discrete logic states in unit-delay time sequence.

A simplification of the switch-level simulator is the unidirectional *gate-level logic simulator*, which uses NOT, AND, OR, NAND, and other combinational logic gates, and state-preserving components such as flip-flops and counters. This simulator solves simple boolean equations to obtain the output state of the logic components. Time may be in unit delay intervals or variable delay, which more closely models continuous time. Unfortunately, not all MOS gate-level elements, specifically pass transistors, are unidirectional in nature, and thus are not suitable for gate-level simulation.

Register-transfer level simulators [Hafer 83, Lewke 83] deal with the overall structure and architecture of a design. Modules, such as full adders and systolic arrays, are specified by procedural descriptions. Because they simulate more abstract modules and their representation of signals is somewhat courser than in the logical case, register-transfer level simulators are usually over an order of magnitude faster than gate-level simulators for the same circuit.

At the highest level of abstraction, *behavioral* or *functional simulators* are used at the initial

design phase to verify the algorithms of the abstract system to be implemented. In contrast to the register-transfer level simulator, the actual structure of the circuit is not necessary for this type of simulation.

2.2 Design Strategy

The first question to answer when developing a new system is *"What are the design goals of our system?"*. The Simulation Environment ties together the simulators needed by all phases of the circuit design process. More specifically, the Simulation Environment in Schema provides (1) simple extensibility for incorporating additional simulators, (2) a foundation for building and integrating new analysis tools, and (3) the capability to perform mixed-mode simulation. These are the major design goals of the Simulation Environment.

A ***uniform interface*** is a natural consequence of the aforementioned design goals. This can be viewed from two perspectives. For the *designer* of CAD software, a uniform CAD interface facilitates additional simulation tools as well as providing the groundwork upon which to build new analysis tools. For the *user* of CAD software, a common interface eliminates the unnecessary task of learning the operation of each individual tool.

The question remaining is *"What approach or design strategy leads to these desired properties?"*. ***Common data representations*** make it possible to create a uniform interface to the user, the simulators, and the analysis tools. The following sections describe the Simulation Environment in Schema, and how this approach achieves the design goals.

2.3 What is a Simulation Environment?

The major components of the Simulation Environment are: a Generic Simulator, common data representations, a single user interface. Figure 2-1 depicts the interactions of each component within Schema. The *Generic Simulator* coordinates the flow of information between the simulation initiator and the individual simulators. The medium for information flow is a *common data representation*, and finally the *user interface* provides a slick graphical interaction with the underlying data structures.

The Generic Simulator acts as an interactive guide in the *Generic Simulation Process*:

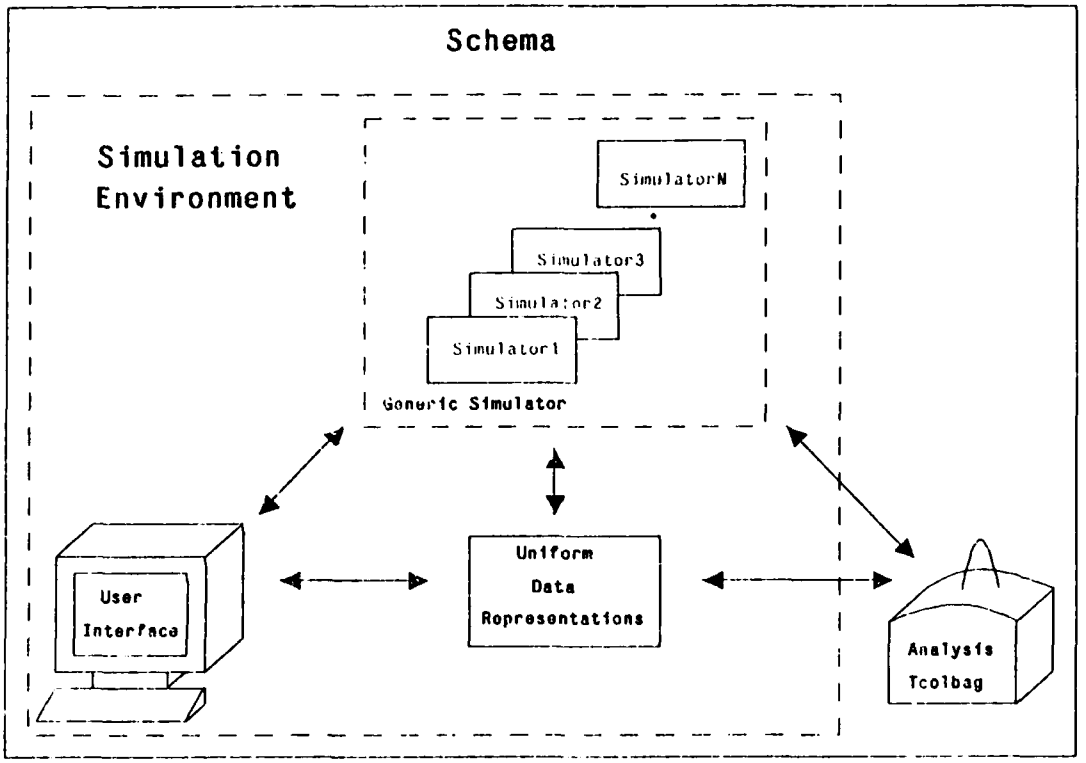


Figure 2-1: Simulation Environment in Schema.

1. The user interacts with the Simulation Environment by way of the user interface. Analysis tools interact directly with the Simulation Environment Once the appropriate input data has been entered, simulation is initiated by the user or by an analysis tool. At this time the initiator chooses a specific simulator from among a rich variety of available simulators and selects a specific region of a circuit for simulation.

2. The Generic Simulator initializes input data for simulation. This may require a translation of the input data to the form required by the selected simulator. Prior to execution the Generic Simulator interactively notifies the initiator in the event of any ambiguities, inconsistencies, or undefined quantities.

3. The simulation is performed.

4. The Generic Simulator interprets the output data and transforms it into a common representation. The results are then presented to the user, again via the user interface, or are made directly available to the analysis tools.

The following sections take a closer look at each component and its role in the development stages of the Simulation Environment. The final section discusses the contribution of each piece toward the design goals.

2.3.1 Common Representation

2.3.1.1 Objects

For electronic simulators, typical *input* data comprise circuit topology, modeling parameters, and excitation signals; typical *output* data are the resulting waveforms. Thus, the basic entities or ***objects*** the Simulation Environment must supply to the Generic Simulator are *circuit topologies*, *models*, and *waveforms*. Determining what objects exist is the first task in designing the Simulation Environment.

For each object to be accepted by a simulator, a corresponding object in the Simulation Environment is defined. Within Schema, a circuit design is made up of components called *modules*. Modules and their interconnections are supplied by the circuit *topology*. Each module may contain some local *model* information. For example, transistor modules may have threshold voltages or logic gates may have propagation delays as part of their model. And finally, signals are the *waveforms* associated with the input to and the output from simulators. In general, these objects represent the data essential for simulation, and thus essential to the Simulation Environment.

2.3.1.2 Object Types

The next task is to further subdivide the types of topology, model, and waveform objects required in the Simulation Environment. *This subdivision is dictated by the* ***types of objects*** *each simulation tool simulates.* A transistor, for example, has a non-linear, linear, and switch model; thus, these model types should be made available in the Simulation Environment. Similarly a circuit-level simulator accepts topological modules including resistors, capacitors, transistors, and waveforms such as exponential or piece-wise linear voltages and currents. The subdivision of topological, model, and waveform types is explored further in Chapter 3, Chapter 4, and Chapter 5, respectively.

There is an overlap in the types of topological, model, and waveform objects accepted by

each simulator. An example of a topological module is the transistor. Although circuit, linear, and switch level simulators all simulate the transistor, it is not necessary to define a different transistor object in the Simulation Environment for each individual simulator that simulates it. *The objects defined in the Simulation Environment are the* ***union*** *of the object types that could possibly be simulated by any of the simulation tools.* This is the key idea behind a common representation for data objects in the Simulation Environment. The user interface, the Generic Simulator, and the analysis tools built into or integrated on top of the Simulation Environment all interact with these uniform data objects.

2.3.1.3 Appropriate Types

Of course, not all objects defined in the Simulation Environment will be accepted by each simulator. A logic level simulator for instance does not simulate capacitor modules, and exponential voltage waveforms. Thus, associated with each simulator is a specific set of *appropriate module*, *model*, and *waveform types*. These represent the types of objects each simulator accepts. Incorporating a new simulator requires the specification of a set of appropriate module, model, and waveform types.

While some simulators handle different types of modules, other simulators share some of the same types of modules. Circuit, linear, and switch-level simulators all have the MOS transistor as an *appropriate module type*. But each of these simulators uses a different model for the MOS transistor module type. A major feature distinguishing one kind of simulator over another is the models it associates with its modules. For any given appropriate module type, there may be one or more *appropriate model types*. For the circuit simulator Spice2, no model is expected for modules of type resistor, yet for the MOS transistor, three model types are possible.

Simulator selection also restricts the *appropriate waveform types*: different signals are required for different simulators. Voltage and current waveforms are expected for a circuit-level analysis, and binary waveforms are required for switch or logic level analysis. To support the mixed-mode capability of the Simulation Environment, if signals of one type can be transformed into another type acceptable to a specific simulator, these types are also part of the simulator's set of appropriate waveform types. If a transformation operation on a binary signal can produce a voltage signal for a circuit simulation, then binary as well as the voltage signals are appropriate signal types for a circuit simulation.

In summary, the object types in the Simulation Environment are the union of the types of objects handled by the different simulators. As every simulator does not accept all types of objects defined in the Simulation Environment, a set of appropriate types are associated with each simulator.[1]

2.3.2 The Generic Simulator

The Generic Simulator is made up of many simulators, and treats each component simulator as a *black box*. It is only responsible for supplying input to and obtaining output from the black box. Thus the Generic Simulator need never know about the internal workings of each component simulator. From outside the Generic Simulator, the user and the analysis tools perceive the Generic Simulator as a black box. Furthermore they never need to interact with the simulators within the Generic Simulator.

The Generic Simulator interactively coordinates the flow of topology, model, and waveform objects between the simulation initiator and each individual simulator. This entails obtaining the input data from the user or analysis tool, supplying the data in the representation required for the simulator, invoking simulation execution, interpreting the resulting output data, and placing the output data into the Simulation Environment for future analysis.

The Generic Simulator interacts with two kinds of simulators: internal and external. An *Internal Simulator* directly manipulates the data objects present within the Simulation Environment, in much the same way an analysis tool built on top of the Simulation Environment would. In this case, the simulation initiator has the opportunity to *interactively control* simulation execution; output signals can be monitored in real time. On the other hand, an *external simulator* creates its own data structures. External simulators typically exist on a remote processor(s) using a separate address space. Computationally intensive simulations are sent off to special-purpose hardware or multiprocessor systems without inhibiting the speed of the Simulation Environment's current process. The combination of internal and external simulation offers the advantages of both strategies and permits a large degree of flexibility in simulation.

The Generic Simulator expects the objects in the Simulation Environment to perform cer-

[1] The sets of appropriate types are not necessarily static. As low level simulation results are summarized into models of more abstract modules, for use in higher-level simulations, the modules and their corresponding models may be appended to the set of appropriate types.

tain tasks, or ***operations***. The operation actually invoked depends on the type of object being asked to perform the operation, yet the object type is irrelevant to the Generic Simulator. The same operation can mean different things depending on the type of the object. This technique is known as *data-directed programming* [Abelson 85]. The following two sections present a more detailed look at both internal and external simulators and what operations are required for each kind of simulation tool.

2.3.2.1 Internal Simulation

Internal simulators have direct access to the objects in the Simulation Environment. Each object involved in the simulation is delegated responsibility for delivering some local information about itself or performing some computation using this information. To do this, specific simulation operations are defined for each appropriate object type handled by the selected simulation routine. For example, the NAND and NOR model types each have their own boolean operation for a logic level simulation.

Internal simulation becomes a layer of these simulation routines where each general algorithm stands alone as an independent, modular unit. Common algorithms could then be shared over different simulators. For example, relaxation-based simulators and asynchronous logic simulators both exploit the inactivity of the circuit by using selective-trace and event-driven algorithms. One routine could serve both simulators. Other generic algorithms are useful for other parts of Schema. The matrix manipulation routines used for the general-purpose circuit simulator may also be useful in handling graphics.

A generic layer of operations on objects would ideally complement this layer of simulation routines. These operations are similarly shared over different simulation algorithms as well as other components of Schema. For instance, most types of waveforms have a generic internal-value operation which calculates and returns a value given a specific point in time. This is a very common operation used not only by circuit level simulators, but also by display routines and analysis tools. One generic operation is defined for each waveform type to satisfy the needs of all potential callers.

2.3.2.2 External Simulation

Prior to an External Simulation, each object in the Simulation Environment requiring simulation must be transformed into the appropriate external representation, usually a textual description language understandable by the simulator. The description is then sent to a separate address space where the simulator builds its own internal data structures for the simulation. If the simulator exists on a remote processor(s), the description is sent via the local network or file system. After simulation execution, the output data must be interpreted and transformed into data objects in the Simulation Environment.

Input transformations are instigated by the Generic Simulator, yet are actually performed by the object itself. As in the Internal Simulation case, the particular operator invoked will depend upon the type of object being transformed. A transistor object requires a very different transformation operator than that of an exponential waveform. Furthermore, because there is exactly one representation for the transistor object in the Simulation Environment and possibly many simulators that use this type of object, there may be many transformation operators defined for it -- potentially one for each external tool that simulates the transistor. A switch-level simulator for example, requires a different transistor representation than a circuit-level simulator and thus a different transformation operator. In the case of output data, the Generic Simulator must however supply a parser to extract the output information and to create the data objects within the Simulation Environment. Transformation responsibility in this case lies with the Generic Simulator.

For both types of simulators, each object has a certain set of operations that it must perform. The Generic Simulator need never know the implementation details of these operations, and each object need not know about the internal workings of the Generic Simulator. The individual simulators, the Generic Simulator, and each object in the Simulation Environment are all perceived as black boxes. Their internal structure and operations are essentially hidden and isolated from each other. *The Generic Simulator can be designed independent of the type of objects it is simulating.* It is *generic* in the true sense of the word. Thus, for the Generic Simulator to perform its task, coordinating the flow of objects within the Simulation Environment, it must simply know what operation to perform and on which object to perform it.

2.3.3 Uniform Interface

The user and the analysis tools interact only with the data objects and the Generic Simulator. Because of the black-box quality of the Generic Simulator, the user and the analysis tools do not interact with the individual simulators.

The analysis tools built on top of the Simulation Environment have *direct* access to the uniform data structures in the Simulation Environment, and thus can interact with the objects in much the same way as an internal simulator. Thus the interface to the topology, model, and waveform objects, as well as the Generic Simulator is simple; the analysis tools need only know the operations defined for each. By just knowing the operations for accessing output waveform objects and the operations for telling the Generic Simulator to halt the simulation process, an analysis tool can interactively monitor the execution of an internal simulation the moment erratic waveform behavior develops.

The user *indirectly* interacts with both the data representations and the Generic Simulator through a graphical interface. The Generic Simulator interface amounts to a well-defined series of textual, or menu-driven commands. The designer is thus spared the burden of learning the operation of each individual simulator; instead, a working knowledge of the Generic Simulator is sufficient. Schematics, layouts, and icons serve as a graphical presentations of the topology. The correspondence between the graphical presentations and the topology is dealt with further in Chapter 3. Models have a simple menu-driven interface. Waveforms have display objects which have the ability to represent themselves graphically to the user; these are discussed in Chapter 5.

2.3.4 Accomplishing the Design Goals

A common representation for data is equivalent to defining a set of object types and a set of operations that can be performed on those types. These types provide the uniform interface which enables us to achieve our design goals. Interfacing new CAD tools requires only local additions to the environment. Integrating an additional simulator may require new object types and a set of operations for each type of object the simulator handles. A new object type is defined for the Simulation Environment only if the simulator actually simulates an object not yet defined in the environment. Adding an internal simulator may also necessitate the modular addition of general simulation algorithms along with some object operations. For an external simulator, a set of transformation operators and an output parser are necessary. Building new analysis tools

requires only a working knowledge of the objects in the environment, the operations that can be performed on them, and the operations that are available for the Generic Simulator. Because waveform objects are represented uniformly in the Simulation Environment, output signals from one simulation can be used as input to another simulation; the mixed-mode property is a direct result of the uniform data structures. With the different levels of simulation, a type transformation operation may be necessary. This is explored further in Chapter 5. In summary, all design goals can be accomplished through the local addition of new objects and operations on those objects.

2.4 Implementation in Schema

The Simulation Environment is implemented in Schema. In this section, a brief overview of Schema's hierarchical organization, constraint network, and creation on demand techniques are all described. In subsequent chapters, we shall see how these strategies tie directly into the Simulation Environment.

2.4.1 Hierarchical Organization

Schema is organized hierarchically as shown in Figure 2-2 where each part in the hierarchy may contain subparts. The root of the hierarchy is the Portfolio which has subparts called *Projects*, and *Environment folders*. Projects serve as an organizational mechanism for grouping together other Projects and Module folders. Environment folders supply the designer with standard libraries. A *Module folder*[2] contains the electronic circuit design; it has *icon*, *layout*, *schematic*, *topology*, and *waveform folder* parts. The user's graphical interface to the topology is mainly through the schematic, layout, and icon presentations. And finally, waveform folders hold collections of waveform specifications, simulation stimuli, and simulation results. This partitioning allows the user to concentrate on one given hierarchical level of design at any particular time. Hierarchical organization is an essential strategy in controlling the complexity of large scale designs.

Each object in the Simulation Environment naturally fits into the hierarchical organization of Schema. The circuit topology and simulation waveforms are parts of Module folders. Because a

[2]Module folders and modules are different entities for historical reasons. They were incorrectly named. Modules are components of the topology, the topology is a component of the module folder.

model may be shared over many modules, models are collected into folders located directly in the user's environment. In later chapters, we shall see how each of the objects also naturally conforms to this hierarchical representation.

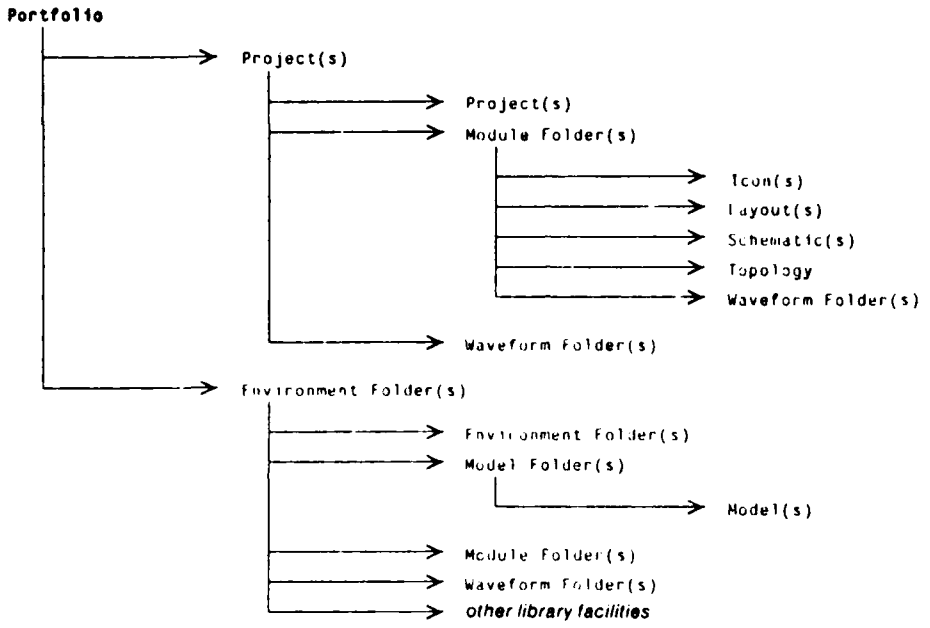


Figure 2-2: Hierarchical organization of Schema.

2.4.2 Constraint Network

Objects may contain *parameters*. Relationships called *constraints* are held between these parameters. A transistor has local width, length, and shape-factor parameters where the width is constrained to be the length multiplied by the shape factor. All constraint relationships are specified in a global *constraint network*. This permits constraints between the parameters of different objects. Complex timing relationships between the parameters of many different waveforms can be captured in the constraint network.

This technique is primarily useful for the *automatic propagation of constraints* through local computation. Modifying one waveform's parameter automatically propagates to those waveforms constrained to it. In the event of far-reaching effects, constraint propagation saves the designer from the tedious and time-consuming process of manual updates. Analysis, synthesis, and reasoning tools can also make use of the constraint network in transistor sizing or circuit verification, for example.

2.4.3 Creation on Demand

Creation on demand is the technique of creating an object's internal structure only when it is needed. In the meantime, the external environment only knows the object exists; typically this is done by knowing the name of the object. Creation on demand applies equally over all objects in the hierarchy. Once the internal data structure has been created, its internal parts likewise need not be created until required. For example, if the designer is interested in only in one specific module folder in a large hierarchy of projects and module folders, then it is only necessary to create the parents of the desired module, beginning with the designer's Portfolio. This technique has the advantage of saving valuable memory space and subsequent garbage collection time - a substantial savings when dealing with large-scale designs.

2.5 Summary

The Simulation Environment provides a uniform CAD interface, a consistent user interface, and mixed-mode capability by using a common representation for simulation data objects: circuit topologies, models, and waveforms. The Generic Simulator coordinates the flow of these objects between each simulator and the simulation initiator. The data objects, the Generic Simulator, and the user interface together make up the Simulation Environment as implemented in Schema.

Chapter Three

Topology

The *topology* contains the interconnection information of a circuit design. The structure of a topology deviates from the general hierarchical organization of Schema in that it does not contain subparts. Instead, the topology has a *module definition* and a *module type*. The *module definition* is used for the simulation of the current topology. The first half of the chapter concentrates on the module definition: its uniform representation, submodule interconnection, and user interface. The module definition defines a new *module type*. The basic module types as well as techniques for creating new module types and operations are examined in the remaining half of the chapter.

3.1 Module Definition

3.1.1 Uniform Representation

The module definition contains submodules, pins, nodes, parameters, and models. The submodules may also have submodules. In this way, modules fit naturally into the hierarchical organization of Schema, as shown in Figure 3-1. Together, the submodules, pins, and nodes specify the electrical connectivity information. Parameters name quantities which are tied to Schema's constraint network. Models are discussed in detail in Chapter 4. The topology, as all objects in the Simulation Environment, has a uniform representation.

3.1.2 Module Interconnection

Module interconnection, an essential piece of electrical information, is accomplished with *pins*, *nodes*, and *global pins*. A *pin* is a module's interface to the outside world. Transistor modules for example contain four pins: gate, source, drain, and body. Modules are interconnected by attaching their pins to *nodes*. And finally, a *global pin* is a special pin seen by all modules spanning the hierarchy. It may connect through a common node to any module pin. Global pins are used mainly for supply voltages such as Vdd and Vss.

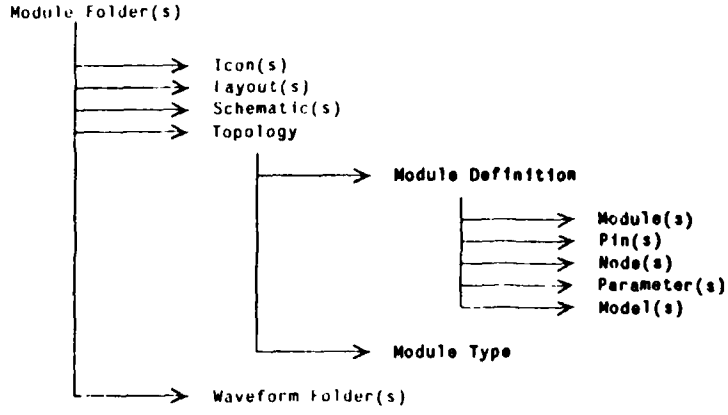


Figure 3-1: The topology and its placement in the hierarchical organization of Schema.

Each module *pin* knows (1) the nodes connected to internal modules, *inodes*, (2) the nodes to which external modules connect, *enodes*, (3) its direction, and (4) its parent module. In Figure 3-2, the inverter module has four pins associated with it: A, A-bar, Vdd, and Vss - of which the latter two are global pins. They all have nodes that connect to the pins of internal modules. Pin A has an internal node n3, no nodes connected externally, the direction *input*, and a parent, the inverter module. The gate pin of the enhancement mode eMOS module has no nodes internally connected, but does have an external node n3, the direction, *input*, and the eMOS module as a parent.

Each *node* knows all the pins attached to it and the internal pins for which it is the internal node. Node n2 is attached to pin A-bar of the inverter module, the gate and drain pins of the depletion mode dMOS module, and the drain pin of the eMOS module. Pin A-bar is the internal pin for which n2 is the internal node.

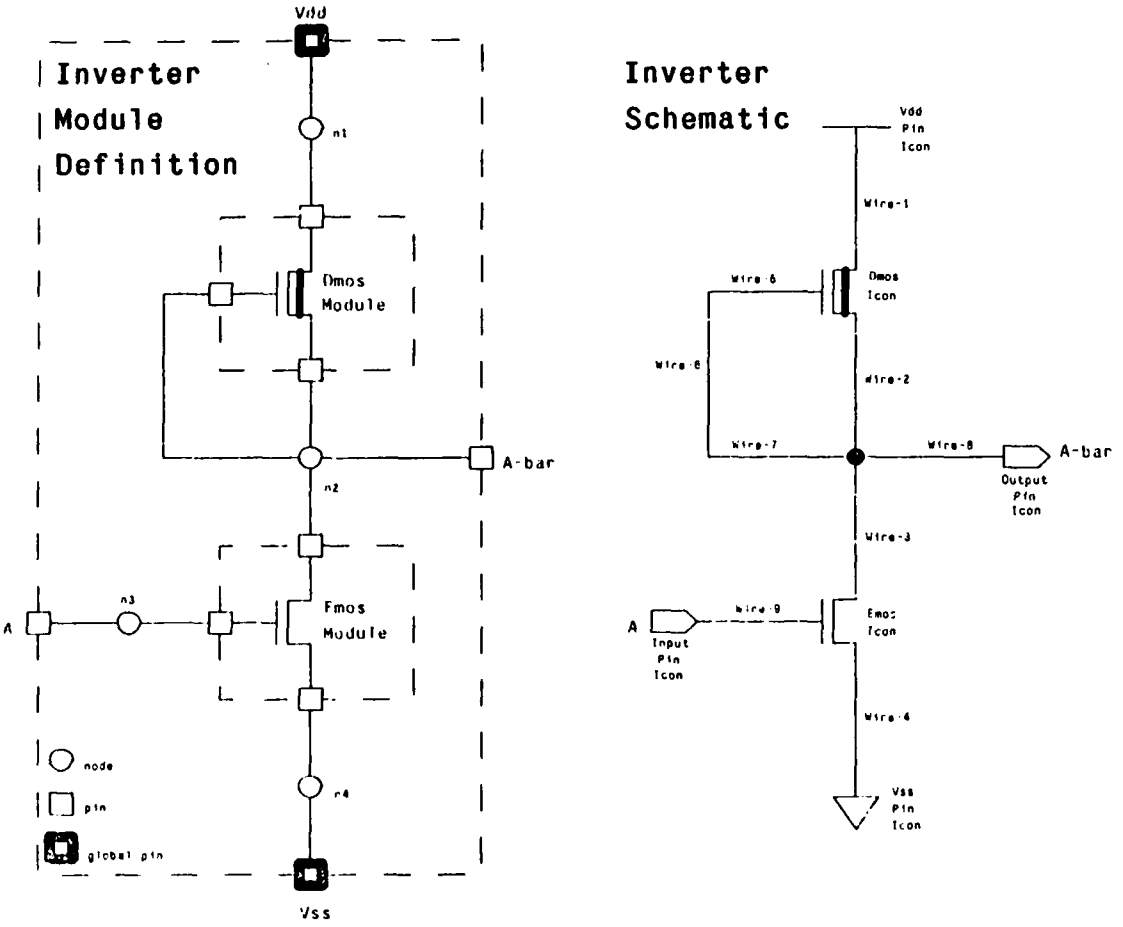


Figure 3-2: Inverter module definition and corresponding schematic presentation.

3.1.3 Module Definition Creation

Prior to initiating a simulation, a module definition must be available. If the definition does not exist, it is initially created from the most recent graphical schematic or layout presentation. A module definition may however already exist from some other simulation. In this case, if it is not up to date with the latest version of the presentation, it is updated. This section describes the process of creating or updating a module definition from the presentation.

The presentation is given responsibility for creating or updating the module definition. If the definition is nonexistent, a dummy module object is created for the definition; it initially has no submodules, pins, or nodes. Then for each part in the presentation, a *topological correspondent* is created in the module definition, if none exists. Topological correspondents are submodules, pins, or nodes in the module definition; the module definition is updated accordingly.

A schematic presentation for example, is composed of icons and wires that contain placement and display information. Pin icons, module icons, and wires in the schematic have topological correspondents of pins, modules, and nodes respectively, in the module definition. A schematic for the inverter is shown in Figure 3-2. `Wire-2`, `Wire-3`, `Wire-5`, `Wire-6`, `Wire-7`, and `Wire-8` of the inverter schematic all have node `n2` of the inverter definition as their topological correspondent. The `Input Pin Icon` and `eMOS Icon` have topological correspondents of Pin A, and the eMOS module, respectively. Associated with each icon is a set of display pins used to connect wires. These display pins are not shown graphically, yet they do have topological correspondents in the module definition. The `eMOS-Icon` has three display pins, each of which has a topological correspondent - the gate, source, and drain pins.

The module definition is created at the top level; the submodules and their interconnections are created. The internal structure of each submodule is only created on demand. Once this top-level module definition has been generated, it may be saved in a topology save file for future use. When the file is read in during a new Schema session, the module definition is not created, but rather a new module type is defined. In this case, it is not necessary to create the definition from the presentation; the definition can be simply created from the module type.

3.1.4 Uniform User Interface

The module definition is visually transparent to the user. The user indirectly communicates with the objects in the topology's module definition via the graphical schematic or layout presentation. During the simulation process, the presentation is used as a *read-only* medium for extracting or modifying electrical information in the module definition. Because each display object has a topological correspondent in the module definition, the user can easily access electrical information. Similarly, each part in the module definition has a presentation correspondent. In this way, the parts of the module definition may report back to the user.

The presentation is a flat structure, whereas the topology is hierarchical. The correspondence between the module definition and the presentation is only for the top-level modules in the hierarchical definition. This presents two problems when the user tries to examine the electrical information in the lower level modules. First of all, the only topological components accessible to the user are those having a correspondent in the presentation. Any parts of submodules in the module definition do not have presentations associated with them. Secondly, these parts may not even exist. When the module definition is first created, only the top level objects and their interconnections may exist.

These problems are solved with the *zoom-in* facility. Suppose an inverter icon is a part of the user's current presentation, and the user wishes to set the length and width parameters of the transistors inside the inverter module. Further suppose the inverter module is not fully created, *i.e.*, the transistors do not yet exist. The zoom-in facility finds the layout or schematic presentation from the module folder of the inverter icon, and makes it visible to the user as a read-only reference for examining the submodules of the inverter module. In order to examine the transistor submodules of the inverter, the inverter must first create its submodules. During the creation process, a correspondence is set up between the inverter's presentation and the module instance in the same manner as before. The advantage to this strategy is a single schematic or layout presentation is useful for all modules of the same type - not just the module definition.

3.2 Defining New Module Types

A *new module type* is defined from the module definition, or from a textual description stored in the topology's save file. The type is used to create a separate copy of the module definition for use as a part in some other module. When the type is created, operations are automatically defined to enable an object of the new *module type* to create its own parameters, constraints, submodules, pins, and internal interconnections. Three basic module types are available: simple, compound, and abstract.

3.2.1 Simple Modules

Simple Module Types do not contain submodules. They may, however, have pins, parameters. and constraint relationships, which are generated as soon as an object of this type is created. Examples of simple modules include the `resistor`, `capacitor`, dMOS and eMOS transistors, and inverter, NAND, NOR, XOR, OR, and AND logic gates. These types are mainly defined in the designer's environment. Another distinguishing feature of simple modules is they typically have no schematic, only an icon. The following examples depict simple module type definition for the `resistor` and eMOS transistor.

```
(defmodule resistor simple
   (resistance)                ;parameter definition
   (pins p+ p-))               ;pin definitions

(defmodule eMOS simple
   (width length shape           ;parameter definitions
    source-area source-perimeter
    drain-area drain-perimeter)
   (pins gate t1 t2 body)
   (c* (>> width)              ;constraint between parameters
       (>> shape)
       (> length)))
```

3.2.2 Compound Modules

Compound Module Types have submodules; and thus, can be hierarchically structured. As with simple modules, pins, parameters, and constraints are all generated when an object of this type is first created. Pin creation is particularly important at this point; external modules can then connect to this module without knowing the internal structure of the module. The submodules and their internal interconnections are created only upon demand. The type associated with each

user-defined topology is usually a compound module type. An example of an inverter module type follows:

```
(defmodule inverter general
   ()
   (global-pins Vdd Vss)
   (pin a input)
   (pin a-bar output)
   (module pulldown eMOS)      ;submodule definitions
   (module pullup dMOS)
   (connect (>> t2 pullup)     ;internal connections
            (>> Vdd))
   (connect (>> t1 pulldown)
            (>> Vss))
   (connect (>> gate pulldown)
            (>> a-bar))
   (connect (>> t1 pullup)
            (>> gate pullup)
            (>> t2 pulldown)
            (>> a-bar))
```

3.2.3 Abstract Modules

Abstract Module Types are generalizations of a class of module types with similar characteristics. For example, there are many module types that have two pins, such as the `resistor`, `capacitor`, and `inverter`. The abstract module type, `Two-Pin-Device`, captures this notion.

```
(defmodule two-pin-device abstract
   ()                          ;no parameters
   (pins p+ p-))               ;pin definition
```

The `resistor` can now *inherit this abstract type, and thus implicitly includes two pins.* This is known as *type inheritance*. The previously-defined simple module type, `resistor`, is redefined as follows.

```
(defmodule resistor simple
   (resistance)
   (includes two-pin-device)) ;inherits two pins
```

Another abstract module, `MOS`, captures the general characteristics of MOS transistors including width, length, and shape parameters. Additionally, a constraint is placed between these parameters.

```
(defmodule MOS abstract
   (width length shape
    source-area source-perimeter
    drain-area drain-perimeter)
   (pins gate t1 t2 body)
   (c* (>> width)
       (>> shape)
       (>> length)))
```

This abstract module is then used to define specific types of transistors, such as eMOS and dMOS, with these implicit parameters and constraints. Type inheritance greatly simplifies the type definition.

```
(defmodule eMOS simple
   ()
   (includes MOS))              ;inherits MOS characteristics
```

3.3 Defining New Module Operations

A layer of general, all-purpose accessors and operations is currently defined for topological objects. This layer is independent of any particular simulator and thus is useful not only to the Generic Simulator, but to any tool requiring access to topological information. One very basic operation gives modules the ability to create their own submodules if they have not yet been created. Another operation permits a module definition to dump its data structure in such a way that a module type is defined when the dump forms are evaluated. Other localized operations may be easily incorporated.

Because a general layer of operations on topological objects currently exists, integrating additional internal simulators does not require the addition of a new operators. For an external simulator, however, a transformation operation must be defined to translate the data objects in the environment into a textual description for the simulator. For each module type the simulator accepts, a new transformation operation is defined. Simple transformation operations for creating a Spice2 input deck are shown below.

```
(defmethod (resistor :spice-deck) (stream)
   (format stream "R~D ~D ~D ~F~%"
           (simulation-resistor-number self)
           (simulation-node-number (>> p+))
           (simulation-node-number (>> p-))
           (>> resistance)))

(defmethod (MOS :spice-deck) (stream)
   (format stream "M~D ~D ~D ~D ~D ~A W=~D L=~D~%"
          (simulation-MOS-number self)
          (simulation-node-number (>> t2))
          (simulation-node-number (>> gate))
          (simulation-node-number (>> t1))
          (simulation-node-number (>> body))
          (send (send self :get-model) :name)
          (>> width)
          (>> length))
```

Notice a single operation is defined for a whole class of MOS devices. In other words, this operation is performed on all modules that have the abstract MOS type; this includes eMOS module type redefined above. Thus, not only is the type inherited, but the operations defined on the type are also inherited.

3.4 Summary

A topology contains a module definition and a type. The module definition is the topological object used in simulation. It is uniformly represented within the hierarchical organization of Schema. It is initially created from a presentation and serves to define the module type. In this way, new types and their operators can be easily integrated into the Simulation Environment. The simple addition of new types and their operators facilitates extensibility to both Internal and external simulators.

Chapter Four

Models

Because simulators *model* the behavior of real devices, models play a vital role in the simulation of circuit designs. In the Simulation Environment, a model may be associated with each module being simulated. Models contain many of the electrical quantities required in simulation. The uniform representation, the user interface, and the basic types of models are all discussed in this chapter.

4.1 Uniform Representation

Models are not hierarchical; they do not contain other models. Instead, models have parameters such as threshold voltages and oxide thicknesses for circuit level transistor models, and setup times, propagation delays and hold times for logic-level models. These parameters are not the associated with the constraint network. In the hierarchical organization of Schema, models applicable to a particular type of module are collected into a *model folder*. Similarly, model folders for different modules are grouped into *environment folders* as shown in Figure 4-1. At any one level in the environment folder hierarchy, there is at most one model folder for each module type.

It is interesting to note that model folders and their respective models are kept separate from the module folder for which apply. Rather models and model folders are classified by environment, and the information contained in the module folder is shared over all the environments. In this way, environments can be configured by a particular fabrication process, for example. By simply switching environments, a new set of models corresponding to a different fabrication process can be used. The major advantage to this approach is that circuits can be designed independent of the fabrication process, or indeed, any other technological division.

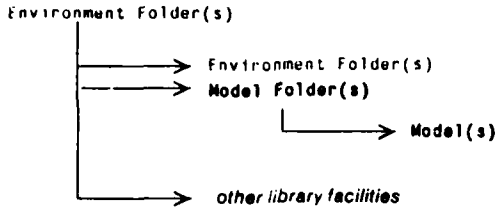


Figure 4-1: Models and their placement in the hierarchical organization of Schema.

4.2 Uniform User Interface

The user interface to creating new model folders and models is simply menu-driven and self-explanatory. If the model folder for the module to be modeled does not exist, a new module folder is first created. A new model is generated by selecting any one of the currently defined model types for the chosen module type. Furthermore, the user is free to modify any of the parameters of the newly-created model.

4.3 Defining New Model Types

In the Simulation Environment, each newly-defined model type must specify both a module type for which it is applicable and a list of parameters. A default value, a short documentation string, and a dimension accompany each parameter definition.

While a model type corresponds to exactly one module type, each module type may correspond to several different models. The MOS transistor is a prime example of a module having many model types: switch, linear, shichman and hodges, analytical, and semi-empirical models. Each model type may produce several individual model objects. There may, for example, be special models for worst-case speed, worst-case power, and worst-case noise margin.

Two basic types of models exist: *models without state* and *models with state*. Models

without state may be shared by modules of a common type, but models with state may not be shared. Modules may require the use of both kinds of models; some parameters may be shared over many devices of the same type, whereas other parameters refer to the local state of the device[3]. The following two sections describe each model type and explain how to define new model types.

4.3.1 Models Without State

Modules of the same type share a common model without state. The obvious advantage to this approach is a savings in memory space because only one copy of the model is generated. This does not imply that *all* devices of a common type must share the same model. This mechanism just facilitates a sharing of a common model. Some modules of a common type may require one shared model without state, while others of the same type may require a different model without state.

Models without state are useful to both external simulators and internal simulators. In a logic level simulation, all NAND gates in the circuit may share common values for transition times along with a common boolean operation. In this case, a single shared model without state is useful to all modules of type NAND, regardless of whether the logic level simulator is an external or internal simulator.

For the abstract module type MOS defined in Chapter 3, a abstract Spice2 model is defined as follows:

```
(define-model MOS spice-MOS ()
  (vt0    0.0    "Zero bias threshold voltage" :voltage)
  (kp     2.0e-5 "Transconductance" :current-per-voltage-squared)
  (gamma  0.0    "Bulk threshold parameter" :sqrt-voltage)
  (phi    0.6    "Surface potential" :voltage)
    ...)
```

The new model type is called `spice-MOS` and its parameters are those that are used over all three MOS device models defined in Spice2. A `spice-MOS-analytical` model type can now be defined with the additional parameters required for simulating an analytical model. Since this new model includes the `spice-MOS` model, all of its parameters will also be included.

---

[3]This case has not yet been dealt with explicitly. Either the two separate models could both be cached in the module, or another type could be defined having local state along with a pointer to the shared model.

```
(define-model MOS spice-MOS-analytical (spice-MOS)
   (lambda 0.0   "Channel length modulation" :inverse-voltage)
   (ucrit  1.0e4 "Crit field mobility degrad" :voltage-per-length)
      ...)
```

And finally, this abstract model is used to define a general model for the eMOS module. The model restricts the channel type to `n-channel`, while also including all the abstract characteristics of the `spice-MOS-analytical` and `spice-MOS` model types.

```
(define-model eMOS spice-eMOS-analytical (spice-MOS-analytical)
   (channel-type "Channel-type" :value nMOS))
```

4.3.2 Models With State

As the name implies, a model with state stores information relating to the current state of the module, such as charge, binary state, and local variable bindings. Internal simulators use models with state to temporarily store simulation data. The `Q` parameter of the `logic-D-flip-flop` model and the `state` parameter of the `Rsim-MOS` model are recalculated for each event or clock cycle of the simulator.

```
(define-model-with-state D-flip-flop logic-D-flip-flop
   (Q           "Current state" :values '(l H X)))

(define-model-with-state MOS Rsim-MOS ()
   (state       "Current state" :values '(on off unknown weak))
   (rstatic-min "Minimum static resistance" :resistance)
   (rdynlow     "Dynamic low resistance" :resistance)
     ...)
```

An implementation of Rsim also requires an initial determination of the effective static and dynamic resistances of each MOS device. These parameters are calculated one time only from the local parameters of each module and are reused over many simulations. To sum up, the parameters of a model with state may depend on the model's local state and the module's local properties.

4.4 Defining New Model Operations

Defining operations for models is a very powerful tool for promoting modularity in internal simulation design as well as in integrating additional external simulators. For internal analysis tools, models perform certain operations such a drain current calculations, boolean functions,

and behavioral-level procedures. For external simulators, transformation operations can be defined similar to those defined on modules.

4.5 Summary

Models have parameters which hold the electrical information required during simulation. Models are located in the designer's environment and are cached in the module prior to simulation. The cached model is then available for future simulations. Two basic types of models exist in the Simulation Environment: models with and without state. New models and operations can be built out of these basic types.

Chapter Five

Waveforms

Waveforms embody any type of excitation or response signal used in the simulation and analysis of electronic circuits. In the Simulation Environment, the uniform waveform representations are patterned after the input and output signals of simulators. This chapter briefly examines these uniform representations and how they fit into the overall hierarchical framework of Schema. Then an introduction to the uniform user interface leads naturally into a discussion of the display types, their associated waveform types and operations, and the usefulness of the constraint mechanism. And finally, type conversions are discussed with respect to the mixed-mode property of the Simulation Environment.

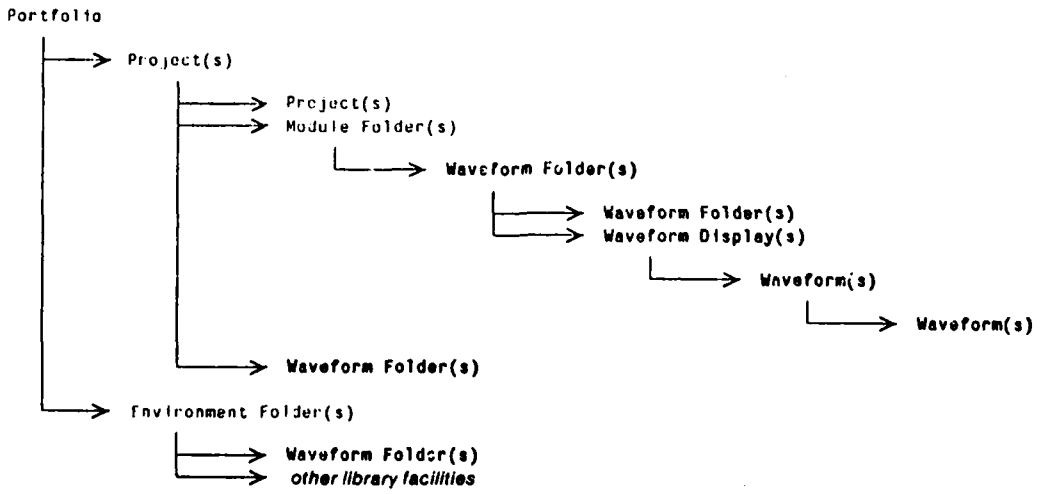


Figure 5-1: Waveforms in the hierarchical organization of Schema.

5.1 Uniform Representation

Waveforms are the uniform mechanism for communication among modules in the Simulation Environment. The means of organizing and grouping waveforms, *waveform folders*, the means of displaying waveforms, *waveform displays*, and the actual *waveforms objects* themselves, provide the mechanism for fitting waveforms into the hierarchical organization of Schema as shown in Figure 5-1. This section gives a brief overview of each, along with its dedicated purpose in the Simulation Environment. This background, in combination with a discussion on the applicability of the constraint network in the waveform domain, lays the foundation for the implementation details presented in the remaining sections.

In the hierarchy of Schema, *waveform folders* are parts of projects, module folders, and environment folders. As a project part, a waveform folder serves as a medium for capturing many of the simulation stimuli, *e.g.*, clocks, control signals, and waveform specifications that are shared between the simulations of different modules. As a module-folder part, a waveform folder contains waveform information pertaining just to the module. Waveform folders that are project and module parts are generic and thus may be shared by many different environments. And finally, as an environment folder part, a waveform folder holds simulation results. In the same way that models are associated with a particular environment, so are the waveforms resulting from simulations that use those models. Allowing waveform folders at many levels in the hierarchy permits a large degree of modularity in organizing the waveforms of very large circuit designs.

Waveform folders contain other waveform folders as well as *waveform displays* as parts. As the name implies, a waveform display object holds the information required for a visual display to the user. A display object, for example, could contain information regarding maximum and minimum axis amplitudes, horizontal and vertical scaling, and dimensional units. This information is conveniently useful to display routines defined for the objects.

One level deeper in the hierarchy, waveform displays hold a ordered set of *waveform* parts. These parts represent the actual signal values. In keeping with the hierarchical structure of parts, waveforms may also have waveform parts. Waveforms are ordered in increasing value along the x-axis to guarantee fast searching through parts.

Constraints may be placed among parameters internal to a waveform, between the waveform parts of a common display object, or across waveform parts of different display objects.

A ramp has parameters of `initial-x`, `final-x`, and `delta-x`. In this case, `delta-x` is numerically constrained to be equal to the difference between the `final-x` and the `initial-x` parameters. This is an example of a constraint placed on parameters *internal* to a waveform.

Another constraint may be tied *between* parameters of waveform parts in a common display object. In a sequence of ramps, the `initial-x` parameter of each ramp part of a display object is constrained to be equal to the `final-x` of waveform part preceding it. This constraint, in conjunction with the aforementioned internal constraint imposed upon each individual ramp, makes it possible to achieve simple shifting operations along the *x*-axis. Changing one parameter locally propagates the constraints to shift all waveform parts of the display object to the right or left along the *x*-axis.

Finally, constraints may be placed *across* waveforms parameters in different display objects. This is especially valuable when specifying complicated timing relationships between input signals. Consider a typical dynamic random-access memory chip where read, early-write, write, read-write/read-modify-write, page-mode read, page-mode write, and Ras-bar-only refresh cycle timing relationships each occupy a full page in the standard MOS memory data book. Local constraint propagation to achieve global consistency over the numerous complicated timing relationships associated with very large performance circuits is a very valuable asset to the circuit designer of today.

5.2 Uniform User Interface

Waveform displays provide a powerful user interface to all waveform objects of the Simulation Environment. They contain the essential data and operations for graphical entry and screen display. The types of display objects defined in the Simulation Environment are geared toward the visual representation universally sketched by today's circuit designers and typically observed on standard test equipment such as the oscilloscope or logic analyzer. Rather than inexact sketching with paper and pencil, complex adjustments of knobs and buttons, and reams of computer simulation printouts, a simple uniform menu-driven, bucky-key interface to each display type is furnished. The user may then graphically enter input waveforms, and view simulation results via a common waveform display interface.

5.3 Display Types and Waveform Types

Display types are selected on the basis of input and output waveform needs for the different simulators. The following sections present a few of the possible types of waveform displays. For each display type, a set of basic waveform types is also defined. Waveform objects are created from these basic types and subsequently become parts of the display objects. Other waveforms can be added to this basic set as long as they supply the necessary graphical entry and screen display routines. Alternatively, additional compound waveform types can be generated from this basic set. This generation of new waveform types is performed in much the same way as the topology's type is automatically generated from the module definition as described in Chapter 3.

5.3.1 Analog Waveforms

Graphically, *analog display objects* are two-dimensional. Horizontal and vertical axes constitute any continuous dimensions, such as voltage, current, time, frequency, power, and capacitance. Maximum and minimum axis amplitudes are also display attributes.

All waveform parts of analog display objects are implicitly given parameters of `initial-x`, `final-x` and `delta-x`, where `delta-x` is numerically constrained to be equal to the difference between the `final-x` and the `initial-x` parameters. The user has explicit control over setting and constraining these values.

Two basic types of waveforms are parts of analog display objects: *functions* and *analog arrays* of (x,y) pairs. Functional types are convenient in three important ways: first as input to circuit level simulators, secondly as a simple graphical entry form for the user, and finally as a compact description of the waveform. Levels, ramps, sinusoids, and exponentials represent the common set of functional types currently available in the Simulation Environment. In general any function, $y = f(x)$, can be included. All functional constants, such the frequency and amplitude of a sinusoid or the time constant of an exponential, are parameters and thus may be constrained. A level waveform type is defined as follows:

```
(defwaveform level simple
   (y))
```

In addition to the implicit parameters and constraint, an explicit parameter, y, has also been defined. In the following type definition, the ramp imposes an explicit constraint between the y parameters; this is similar to the implicit constraint imposed in the x-direction.

```
(defwaveform ramp  simple
   (initial-y final-y delta-y)
   (c+ (>> final-y)
       (>> initial-y)
       (>> delta-y)))
```

Because simulation output of circuit level simulators is typically long listings of (*time, value*) pairs, an array waveform type is the most efficient data structure for memory storage. A summarized `graphical-form` is created to allow for fast visual display.

```
(defwaveform analog-array simple
   (pts graphical-form accuracy))
```

Frequently the output points resulting from a detailed simulation run are extraneous. Furthermore, the designer is often only interested in a transition time or time constant of some selected portion of the waveform. At the expense of some accuracy, many of the points are discarded and replaced with a summarized version. In essence, this summarization process can be viewed as a conversion between the waveform array type and the functional waveform type. At first, the array waveform could be naively viewed as a series of ramps. At this point, the major difference between the two types is the inherent constraint mechanism associated with the ramps parameters. One-to-one mapping for the conversion of the detailed array to the ramp type would be absurd. A more realistic approach applies a combination of heuristic techniques, rigorous curve-fitting algorithms, and desired accuracy level to produce a summarized series of piecewise-linear segments, or a combination of piecewise linear and exponential segments, as is more typical of waveforms resulting from a digital circuit. Detailed simulation results are then discarded for the more summarized version. The currently defined conversion operations are presented in [Solden 86].

In addition to conversion and summarization operations on functional and array waveform types, many mathematical operations are defined [Solden 86]. Standard *unary* operations useful in analysis are interpolation, differentiation, and integration. Others standard operations involving more than one waveform operand include addition, subtraction, multiplication, and division. Operations such as these are extremely powerful for calculating power lossage, effective resistance and capacitance. Moreover, defining additional waveform operations is simple. It requires only a local understanding of the waveform data structures described above, in addition to knowledge of the basic operations already defined.

5.3.2 Binary Waveforms

A *binary display type* is built on top of the analog display type with a restriction placed on the maximum (*1*) and minimum (*0*) amplitude of the *y*-axis. The *x*-dimension is either continuous (variable-delay) or discrete (unit-delay) time.

Three basic types of waveforms can be parts of binary display objects: *steady-state*, *transition*, and *binary array*. These types were selected on the basis of their usefulness as input and output to linear model, switch, and logic level simulators.

Steady-state and transition waveforms implicitly inherit the same parameters and constraint in the x-direction described for the analog case. In addition, steady-state waveforms have a state parameter, and transitions have `initial-state` and `final-state` parameters. States may have values of logic zero (*0*), logic one (*1*), a high-impedance (*Z*), and an unknown (*X*).

```
(defwaveform steady-state simple
   (state))

(defwaveform transition simple
   (initial-state final-state))
```

Steady-state and transition waveforms are similar to the level and ramp defined for analog displays, yet with the restriction on values of state. In the case of the transition however, a constraint was not placed between the `initial-state` and `final-state` as was done between the `start-y`, `end-y`, and `delta-y` for the ramp because `delta-y` would always be either *1* or *-1*.

As in the analog case, binary arrays are a condensed form of output storage. Points are restricted to be (*x,state*) pairs.

```
(defwaveform binary-array simple
   (pts))
```

Conversion between steady-state / transition waveforms and binary array waveforms is a straightforward mapping. Boolean operations, bit-pattern searching, and other virtual logic-analyzer operations can be easily incorporated.

5.3.3 Defining New Displays and New Waveform Types

Analog and binary display types are designed to cover most all the cases for the lower level simulators. This listing is by no means exhaustive. For this reason, adding *new waveform types* for these display types is a simple procedure. Infinite as well as periodic waveform definitions could also be added. From the basic set of simple waveform types defined above, a library of compound, hierarchical waveform types can be defined. *New waveform displays* may also be created. A qualitative [Williams 84] display for example could be built on top of the analog display type, incorporating the display procedures currently available in the Simulation Environment. Non-linear and multi-dimensional display axes and graphics routines could be integrated.

At higher levels of signal abstraction, waveform axes are no longer of any use. Waveform displays amount to program descriptions, flow graphs, state diagrams, and the like. Instead of viewing individual binary signals for example, a collection of signals numerically represented in base *8* or *16* would provide the greatest amount of flexibility. *Octal* and *hexadecimal waveform types* would most likely exist where collections of up to *8* and *16* binary display objects, respectively, could be directly mapped.

5.4 Mixed-Mode Capability

In order to perform mixed-mode simulation, where the output results of one simulation are used as the input in some other simulation, a waveform conversion may be necessary. Simple handlers transform waveforms from one type to another on demand. Conversion techniques *among* waveforms occupying analog display objects and binary display objects have been briefly discussed. Conversion *between* analog and binary waveforms is of greater interest for the providing the mixed-mode capability of the Simulation Environment. In general, conversions from the more accurate waveforms to a higher level of abstraction is straightforward. Mapping a voltage waveform onto a binary waveform requires an understanding of the threshold voltages and currents for the different logic states in the chosen technology . In the opposite direction, techniques are available and are documented in the literature [Arnout 78, Antognetti 81]. All coercions could be easily implemented and integrated with little knowledge of the internal workings of the surrounding Simulation Environment.

5.5 Summary

The uniform representations of waveforms, waveform displays, and waveform folders naturally conform to the hierarchy of Schema. Waveform displays provide a uniform interface to the user. Display types and their associated waveform types are designed to satisfy the input and output requirements of simulators. Parameters associated with input waveform tie directly into the constraint network of Schema; output waveform types conserve on memory storage space. New waveform types and operations as well as display types can be easily integrated. Local coercion routines can be defined to simply transform one type of waveform to another; this gives the Simulation Environment the capability to perform mixed-mode simulation.

Chapter Six

Generic Simulator

This chapter explores the *Generic Simulation Process*: a series of steps leading to a single simulation with the Generic Simulator. As the process unfolds, the discussion centers on how the Generic Simulator coordinates the flow of data objects between the simulation initiator, the user or analysis tool, and the selected simulator.

6.1 Uniform User Interface

During the *Presentation Editing Mode*, the user graphically draws a schematic or layout of a circuit design. Then the user enters *Simulation Mode*. The display is reconfigured to provide both a waveform editor and a read-only presentation viewer. The Generic Simulator requests the presentation to create or update the module definition. During this time, a correspondence is set up between the presentation and the module definition. The read-only *presentation viewer* can then serve as the user's interface to the electrical information in the module definition. At this point, only the top-level submodules of the module definition and their interconnections correspond to the flat presentation. The user may at any time access the internal parts of a submodule via the zoom-in feature described in Chapter 3. Using the *waveform editor*, the user may graphically enter new waveform displays as well as view the waveform displays of any currently existing waveform folders. For example, the user may wish to use a waveform folder containing input test vectors and output specification waveforms for an add or memory-write operation. The combination of both the presentation viewer and waveform editor enables the user to assign waveforms to the input nodes and pins of the module definition. After input waveform assignments, the user may begin the Generic Simulation Process.

6.2 Initiation Phase

To begin a *Generic Simulation Process*, the initiator first selects a region or all of a module definition upon which to perform the simulation. Next a specific simulator is chosen from the available simulators within the Generic Simulator. Simulator selection specifies the set of appropriate module, model, and waveform types handled by the simulator.

Because some simulators perform more than one type of analysis, an *analysis context* must also be specified by the initiator. Traditional circuit simulators for example perform dc, ac small-signal, and transient analyses. If more than one analysis type exists for any chosen simulation, analysis context selection may further restrict the appropriate module, model, and waveform types. For example, a dc analysis context greatly simplifies a capacitor or inductor model. Under different analysis contexts, a different kind of signal may be necessary; a dc analysis produces voltage and current values, whereas a transient analysis generates a history of (*time, value*) pairs.

The Generic Simulator may request additional information from the initiator. In the case of a transient analysis for example, initial time, time step size, final time, and number of simulation steps are required information. For an internal simulation, the initiator has the opportunity to control simulation execution, *e.g.*, to halt when certain waveforms fail to meet output specifications, or to supervise some combination of output waveforms such as effective capacitance.

6.3 Initialization Phase

Once a simulation has been initiated, the Generic Simulator initializes as much information for the simulation as possible. This includes locating the appropriate modules, waveforms, models, and the relationship between them. The Generic Simulator passes type-dependent tasks onto each object. All initialization is completely transparent to the initiator. The following sections describe the Generic Simulator's role in the preparation the uniform data objects in the Simulation Environment for simulation execution. This constitutes the *Initialization Phase* of the Generic Simulation Process.

6.3.1 Locating Appropriate Modules

The Generic Simulator locates the appropriate modules for the selected simulator by requesting this information from the module definition. The module definition asks all of its submodules in the selected region to return the modules to be simulated. If a submodule is an appropriate module type, it just returns itself to the Generic Simulator. If the submodule is not appropriate and is a compound module type, it creates its submodules and their interconnections - if not already created from some other simulation - and forwards the request onto its submodules. If a simple module type is encountered which is not appropriate, an error is signaled; the initiator is then notified that the selected simulator is unable to simulate this particular module type. The recursive process continues until all appropriate modules in the selected region are located. In this way, the responsibility for finding the appropriate modules is passed from the Generic Simulator, to the module definition, and onto each submodule.

As an aside, notice that the entire submodule hierarchy need not be fully generated. Submodule creation is required only down to the appropriate modules in the selected region. This results in considerable time and memory savings - especially when simulating very large circuits at higher levels of abstraction. Even though the circuit may be hierarchically defined down to the detailed transistor level, the existence of the lower level objects is unnecessary for the simulation at hand. For example, consider performing a register transfer level simulation of a microprocessor chip, where a programmable logic array PLA is one major component. The module definition of the PLA may have been separately defined and tested at the detailed transistor level, where simulation results were summarized into a more abstract logic level model. For a logic level simulation of the microprocessor chip, valuable memory space is conserved by not creating the internal transistor structure of the PLA submodule. Creation on demand inhibits submodule generation unless absolutely necessary.

6.3.2 Interconnection of Appropriate Modules

Once the appropriate modules have been located, the Generic Simulator determines the interconnections for the selected simulator. Because modules in the Simulation Environment are hierarchical, the pins of appropriate modules are *indirectly* connected to other appropriate modules via the nodes and pins along the hierarchy. Unfortunately most simulators do not handle hierarchically interconnected modules. To solve this problem, the pins of appropriate modules are *directly* interconnected through a common ***simulation node***. Conversely, each pin of an

appropriate module connects to a simulation node. In this way, the Generic Simulator, and thus the selected simulator, may view the circuit as a flat structure of interconnected modules.

6.3.3 Locating Appropriate Waveforms

Input waveforms assigned by the initiator must be of the appropriate signal type for the simulator, and if not, must be transformed into the correct signal type. The Generic Simulator asks each top-level input node or pin of the hierarchical modules in the selected region to return a waveform of the appropriate type for the simulator. Each node or pin forwards the operation onto the attached waveform. If the waveform is not of the correct type, the waveform calls a transformation operation on itself, which returns an appropriate waveform to the Generic Simulator. If the waveform undergoes a type conversion, the transformed waveform is cached on the node or pin from whence in came; now both the original waveform and its transformed counterpart are available on the hierarchical module definition. This avoids unnecessarily repeating the transformation procedure in future simulations.

6.3.4 Attaching Appropriate Waveforms

6.3.4.1 Input Waveforms

The initiator attaches input waveforms to nodes and pins of the hierarchical module definition. Yet the Generic Simulator associates appropriate waveforms with the flat structure of interconnected modules. When an appropriate waveform is returned from a hierarchical node of the module definition the Generic Simulator attaches it to a corresponding simulation node. Voltage, binary, symbolic and other abstract waveforms are associated with simulation nodes. Some simulators however also associate waveforms with pins. Circuit level simulators for example commonly employ current waveforms. In this case the Generic Simulator creates a new set of simulation pins corresponding to the pins in the selected region of the module definition that were assigned input waveforms. These new simulation pins are different from the pins in the hierarchical module definition because they are directly connected to the flat simulation nodes. The Generic Simulator then attaches appropriate waveforms to these simulation pins.

6.3.4.2 Output Waveforms

Output waveforms are placed on the simulation nodes. If output waveforms are also associated with pins, a set of output simulation pins are created for each module to be simulated. As with input pins, output pins are connected directed to the flat simulation nodes. If an internal simulator has been invoked, the output waveform displays are generated and attached to their respective nodes and pins during the initialization phase for pending availability to other Generic Simulation Processes. They act as virtual waveforms and can be assigned as input in other internal simulations, *i.e.*, for concurrent mixed-mode simulation. For external simulators, it is not necessary to create the waveforms until simulation execution is complete. In the event of extensive or lengthy external simulation, creating the waveform displays ahead of time only adds long-term objects to local memory, and needlessly increases memory paging.

The flat structure now contains the modules, their interconnections, and the input waveforms required for the simulator. The appropriate modules and the input waveforms are the exact same objects contained in the hierarchical module definition, yet the simulation nodes, simulation pins and the output waveforms are newly created for each simulation performed. Furthermore, simulation nodes and pins and their associated waveform data are stored *independently* from the hierarchical module definition; no explicit pointers exist from the module definition to the objects in the flat structure. In this way, each simulation run is kept separate and distinguishable from other simulations, and thus can be quickly and easily discarded, saved for later use, or compared against the results of other simulations. Because waveform output often contains many data points, it is important to summarize the essential waveform data and to discard or *garbage collect* the rest.

6.3.4.3 Mapping Waveforms onto Nodes

A mapping table is created relating the flat simulation nodes and the electrically equivalent nodes in the hierarchical circuit module. Interconnected nodes along the hierarchy correspond to one simulation node, and one simulation node maps onto one or more electrically equivalent nodes of the hierarchical module definition, as shown in Figure 6-1. This mapping allows the user and the analysis tools access to the waveform data from the hierarchical module definition, and vice versa. The user, for example, may probe a wire of the graphical presentation for a waveform. The wire forwards the message onto its topological correspondent, a node in the module definition. The mapping table is consulted for the equivalent simulation node. Once found, the simulation node then returns the waveform. In the reverse direction, waveforms can now find wires of

the presentation for which they are associated. Suppose the user is viewing a waveform and wishes to know the wires in the presentation for which a particular waveform applies. The waveform forwards the operation onto its simulation node. Next the mapping table is consulted for the set of electrically equivalent nodes in the hierarchical module definition. With the knowledge of the user's current presentation, a single node is selected. And finally this node requests each of its presentation correspondents, graphical wires, to display themselves to the user.

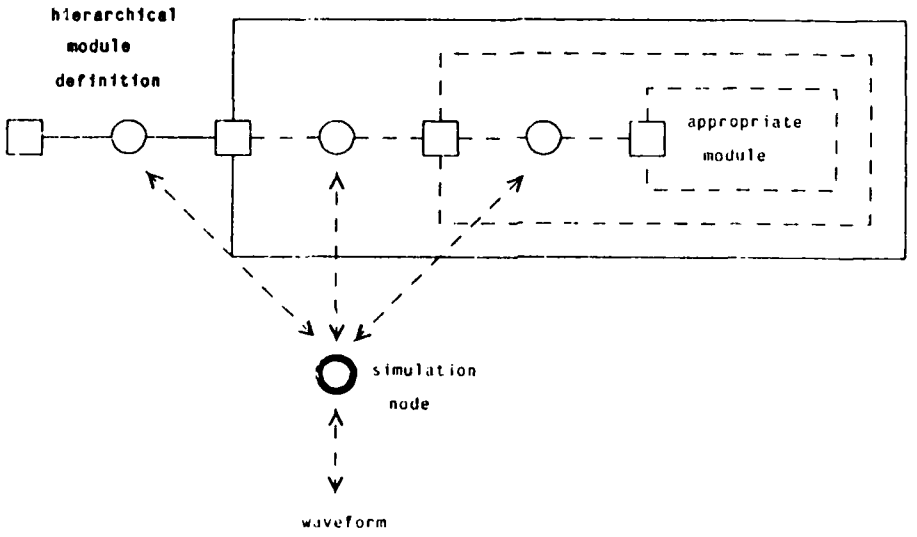


Figure 6-1: Mapping of a single simulation node onto electrically equivalent nodes of a hierarchical module definition.

6.3.4.4 Mapping Waveforms onto Pins

If waveforms are associated with pins, a mapping table for pins is also useful, in this case, a one-to-one mapping. An *input* pin of the hierarchical module definition maps directly onto one simulation pin. *Output* waveforms attached to simulation pins can map onto the pins of the appropriate modules. In contrast to nodes, pins along the hierarchy and their associated currents are not electrically equivalent; pins of hierarchical modules will not have a waveform initially - nor an entry in the mapping table - unless the pin is queried for one.

Suppose the initiator probes for an output current waveform of a module's pin. The pin then consults the mapping table for its corresponding simulation pin containing the waveform. If the module is an appropriate module, an output current waveform is returned. If not, a current waveform must be created for the pin, as shown in Figure 6-2, where a simple application of Kirchoff's current law produces the desired waveform. The current waveform of the hierarchical module's pin is actually the sum of the currents attached to the pins of the interconnected appropriate modules inside *(or outside)*. The procedure is performed as follows. First the hierarchical pin finds all the waveforms internal to its parent module by requesting a current waveform from all internal pins connected to its internal node. If these pins cannot locate a waveform in the mapping table, the request is again forwarded. This recursive process continues until all internal currents have been found. The hierarchical pin then performs a generic add operation on the waveforms returned. This newly generated waveform is then assigned a simulation node and cached in the mapping table for future reference. Generating waveforms only upon inquiry is again part of Schema's creation on demand technique.

6.3.5 Locating Appropriate Models

The Generic Simulator locates an appropriate model, if any, for each appropriate module taking part in the simulation. The model may be found in one of two places. It may already exist within the module itself, cached from a previous simulation, or it may be found in the designer's environment folder. In the latter case, if the appropriate model is of type, model without state, the model itself is cached. If the appropriate model is of type, model with state, a copy of the model is created and cached in the module.

The location procedure occurs as follows. The Generic Simulator simply asks each appropriate module to find a model for the simulator selected under the current analysis context. The module then looks to see if any of its cached models are appropriate. If not, the designer's environment folder is passed the responsibility. Next the environment folder searches through its subparts for a model folder with the correct module type. If not found, each subenvironment is searched, and so on in a breadth first manner[4]. Once found, the model folder is asked to locate an appropriate model. It then searches its models while asking each if it is appropriate. In effect, the responsibility for finding an appropriate model is passed naturally from the Generic Simulator,

[4]This is not currently the case, only one level of the environment folder hierarchy is searched.

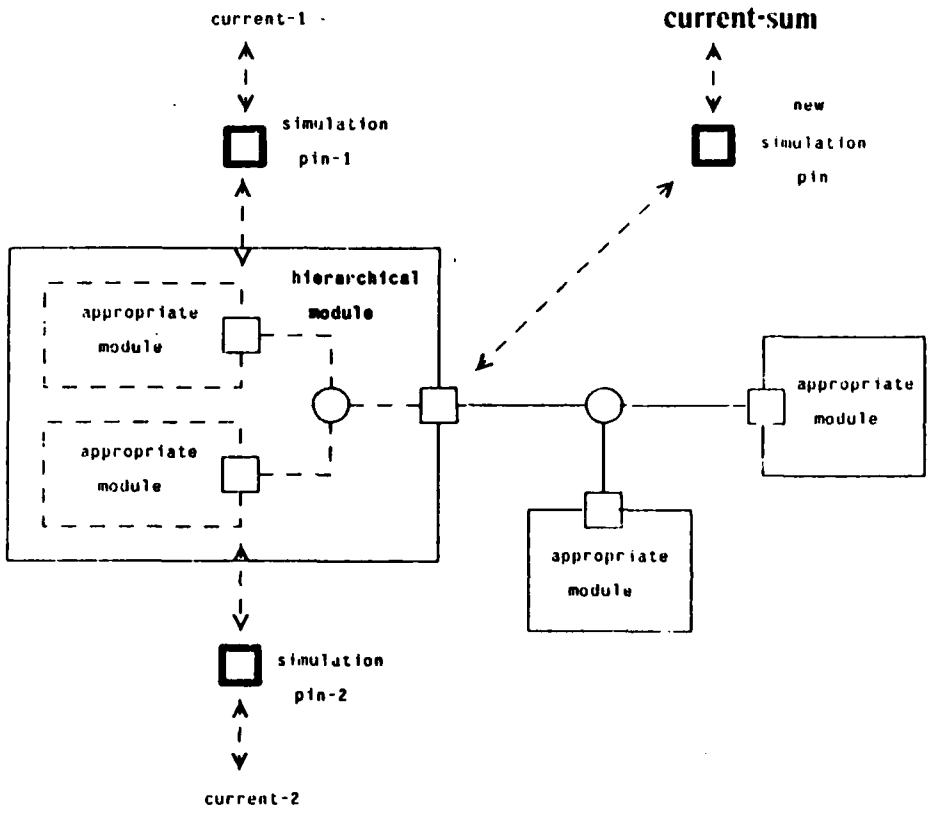


Figure 6-2: Summing current waveforms, *current-1* and *current-2*, to produce *current-sum* for pin of a hierarchical module.

to the module, to the environment folder, to the model folder, and finally onto the model. If the appropriate model is found, either the model or a copy of the model is returned back to the module, and cached for use in future simulations. The Generic Simulator need never know anything about the models.

During the course of simulation initialization, the Generic Simulator notifies the initiator of any inconsistencies, undefined quantities, or ambiguities in the information gathered by the Generic Simulator thus far. Simulation execution cannot proceed until all required information is supplied. The simulation initiator may need to subsequently add or modify waveforms, models, or

parameter values. At the close of the Initialization Phase, data associated with the simulation is locked from modification; all objects however are read-accessible to other processes, including other Generic Simulation Processes.

6.4 Execution Phase

The Generic Simulator handles two basic types of simulators, internal and external. An Internal Simulator directly manipulates the data objects present within the address space of the Simulation Environment. Simulation execution may be interactively controlled. An external simulator generates its own internal data structures in a separate address space. The following sections briefly describe each simulation process and the role of the Generic Simulator in the execution phase.

6.4.1 Internal Simulation

Because an internal simulator accesses the data objects directly, the Generic Simulator need only call the simulation routine and pass it the flat module structure to be simulated. The simulator then forwards many of the type-dependent tasks onto the data objects. In a Spice-like circuit-level simulator, each module and input waveform calculates its fill-in values for the sparse modified-nodal-analysis matrix. Each module's model is responsible for performing calculations based on its model, parameters, and some local state. Input waveforms compute a voltage or current value for a given timepoint. At higher levels of simulation, the simulator dynamically schedules the sequence of operations, or events, as signal values propagate through the circuit. This time the model computes an output waveform value, given some input waveform values. The simulator propagates the calculated output to the input of interconnected modules by way of the flat simulation nodes.

At each time step of execution, input waveforms are sampled, output values are produced and sent directly to the output waveform display objects. Simulation execution can be interactively controlled by the simulation initiator. The user for example can visually observe the output waveforms as the simulation proceeds, and may halt execution in the event of erratic circuit behavior. An analysis tool could dynamically discontinue execution at the moment the resulting waveforms fail to meet design specifications. Not only may the output waveforms themselves be observed, but any combination of operations on these waveforms may also be ob-

served, *e.g.*, power consumption. Furthermore, with the waveform transformation capability of the Simulation Environment, concurrent mixed-mode simulation is also possible. As output waveforms of one region's simulation becomes available, they could be automatically used as input to some other region's simulation.

6.4.2 External Simulation

An external simulation is performed in a separate address space. In the event the simulator exists on a remote processor(s), the Generic Simulator first establishes a connection to the simulation server, typically via a local network. Because more than one remote processor may run the selected simulator, the Generic Simulator polls each of the existing processors to determine the best available resource. Spice2, for example, is highly portable and thus runs on many different servers. Yet at any point in time, some servers may be fully-loaded with performing simulations or some other computationally intensive task. The least-loaded, most efficient machine should be prompted to service the simulation request.

Next the Generic Simulator requests a textual description from all data objects to be simulated. Each appropriate module, waveform, model and parameter object then returns a textual description to be forwarded by the Generic Simulator to the selected simulator. Simulation may then proceed in a background process. During the course of execution, other activities or processes occurring within the Simulation Environment may continue uninterrupted. Upon completion, some textual output is returned. Generic simulator sends the data to an output parsing routine which interprets the output results and creates the uniform waveform data objects in the Simulation Environment. And finally, the simulation initiator is notified of execution completion.

6.5 Completion Phase

During the *Completion Phase*, output waveforms are available for inspection, analysis, and as input to other Generic Simulation Processes. The waveforms are accessible to the initiator via the module definition. In the case of the user, the interface to waveforms attached to the module definition is by way of the presentation viewer in combination with the waveform editor. Convenient analysis tools summarize waveform data for example, not only for graphical display and reduced storage, but also into a new model for use in a higher level simulations as described in Chapter 7.

A Generic Simulation Process may be extended, in which case the same flat structure is reused. The same waveform objects are just appended with additional output points. Output waveforms are collected together into an output waveform folder. Because simulation results are dependent on the models used in the simulation, they are stored with the models in the user's environment folder.

6.6 Summary

The Generic Simulation Process is a series of steps leading to a single simulation on the Generic Simulator. The process occurs as follows. First of all, waveforms are assigned to the input terminals of a circuit module. The simulation initiator, either the user or analysis tool, selects a specific simulator from among a rich variety. The Generic Simulator then prepares the chosen region, the assigned waveforms, the appropriate models, and the module parameters for the selected simulator. Next simulation is performed either directly on the data objects within the Simulation Environment, or externally in a separate address space. Output waveforms are created and made available for inspection, analysis, or as input to future simulations.

Chapter Seven

Discussion

7.1 Summary

The Simulation Environment provides a uniform CAD interface, a single user interface, and mixed-mode capability by using a common representation for simulation data objects: topologies, models, and waveforms. The data objects, a Generic Simulator, and the user interface together make up the Simulation Environment as implemented in Schema.

The object types and corresponding operations defined in the Simulation Environment are patterned after the requirements of the simulators that use them. The addition of new types of objects and their operators facilitates easy extensibility to additional simulators. The object types and the layer of operations defined in the Simulation Environment serve as the foundation upon which to build new analysis tools. Local coercion routines can be defined to simply transform one type of waveform to another; this gives the Simulation Environment the capability to perform mixed-mode simulation.

The Generic Simulator coordinates the flow of objects between each simulator and the simulation initiator, the user or analysis tool, during the Generic Simulation Process. Waveforms are assigned to the input terminals of a circuit module. The simulation initiator selects a specific simulator from among a variety of simulators. The Generic Simulator then prepares the circuit module, the assigned waveforms, the appropriate models, and the module parameters for the selected simulator. Next simulation is performed and finally output waveforms are created and made available for inspection, analysis, or as input to future simulations.

7.2 Implementation: The Simulation Environment Layer

The Simulation Environment is implemented in Schema using Symbolics 3600-family lisp machines. All types are built on top of the Flavor System [Reference 85] provided by the Zetalisp language. The object-oriented programming strategy established by the flavor system provides the base layer upon which Schema is established.

Many of the basic topology, model, and waveform data types and operations have already been defined for the Simulation Environment in Schema. New types and operations are continually being added and refined to conform to the needs of additional tools built into the system. It is hoped that this define-and-refine process will at some point converge to an optimum general representation for data objects, where these representations form a solid layer upon which to build other CAD tools for all areas of circuit design.

Currently two simulators have been implemented in the Simulation Environment; an internal transient simulator and the external circuit-level simulator Spice2. The internal simulator employs the forward-euler method of integrating current into each capacitive node of a circuit. This simulator does not have the accuracy of the detailed circuit analysis simulator, but does have the advantage of being much faster and highly interactive. Thus the designer is able to make initial verification and performance estimates using the interactive internal simulator and save the detailed analysis for the remote simulation engine. Both make use analog waveforms.

The next step is the addition of the linear, switch, and logic-level simulators that use the binary waveforms already defined in the Simulation Environment. For mixed-mode operation, coercion routines between analog and binary waveforms must also be defined. These simulators, together with the currently embedded transient simulators, constitute an essential layer of tools upon which to integrate higher-level simulators.

7.3 Future Work: The Concurrent Mixed-Mode Simulation Layer

Because errors may be introduced into simulation results by an unfortunate choice of simulator at a critical point in the circuit, expert or automatic partitioning routines could be independently developed and placed on top of the Simulation Environment. The routines would essentially divide large scale circuit modules into collections of submodules to be simulated at different levels of abstraction. Critical paths and tightly-coupled subcircuits are grouped and simulated at a detailed level, while less critical circuits are simulated more abstractly.

Concurrent mixed-mode internal simulation is now possible. The Simulation Environment provides the foundation layer of simulators, a Generic Simulator, and uniform representations. On top of this are three essentially independent layers. One provides the signal transformation

procedures for mixed-mode operation, another contains the different internal simulators and general simulation algorithms, and finally the third embodies the expert partitioner. These provide the base upon which to build a concurrent mixed-mode simulator. As waveform values of one subcircuit's simulation become available, they could be used immediately as input to an interconnected module's simulation.

As cited in Chapter 1, the main bottleneck with such a single-system approach is the limited computational power. In Schema, the data objects exist in a common address space with the potential for *multiple processes*. Circuit partitioning conveniently lends itself to parallel processing and could thus spawn off new processes when necessary. Unfortunately however only one processor is currently available. In the future, these processes may be mapped onto more powerful parallel, multi-processor systems. In the meantime, the Simulation Environment provides the foundation upon which to *develop* these more sophisticated software layers.

7.4 Conclusion

This thesis has two main conclusions. First, designing the layer of general representations is the most difficult task in developing the Simulation Environment. Second, once the general representations have been designed for a specific simulation level, it is easy to integrate additional simulators at that same level. In general, as each new simulation level is incorporated into the environment, the representations undergo a continual define-and-refine process. As a consequence, the representations eventually evolve into the most general form satisfying the needs of a comprehensive range of simulators and the needs of the user.

References

[Abelson 85] Abelson, H. and Sussman, G. J., *Structure and Interpretation of Computer Programs*, The MIT Press, 1985.

[Abramovici 83] Abramovici, M., Levendel, Y. H. and Menon, P. R., "A Logic Simulation Machine," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **CAD-2**(2):82-94, April 1983.

[Antognetti 84] Antognetti, P., Pederson, D. O. and de Man, H. (Eds.), *Computer Design Aids for VLSI*, Martinus Nijhoff, 1984.

[Arnold 85] Arnold, J. M., "Parallel Simulation of Digital LSI Circuits," Technical Report 333, Massachusetts Institute of Technology, February 1985.

[Arnout 78] Arnout, G. and de Man, H., "The Use of Threshold Functions and Boolean-Controlled Network Elements for Macromodelling of LSI Circuits," *IEEE Journal of Solid-State Circuits* **SC-13**(6):326-332, June 1978.

[Borrione 83] Borrione, D., Humbert M., Le Faou, C., "Hierarchical Mixed-Mode Simulation Mechanisms in the CASCADE Project," Anceau, F. and Aas E. J. (Ed.), *VLSI '83*, Elsevier Science Publishers B. V., August 16-19 1983, pp. 119-129.

[Bryant 81] Bryant, R. E., "A Switch-Level Simulation Model for Integrated Logic Circuits," Ph.D. Thesis, Massachusetts Institute of Technology, March 1981.

[Chawla 75] Chawla, B. R., Gummel, H. K. and Kozak, P., "MOTIS -- An MOS Timing Simulator," *IEEE Transactions on Circuits and Systems* **CAS-22**(12):901-909, December 1975.

[Chen 84] Chen, C. F., Lo, C., Nham, H. N. and Subramaniam, P., "The Second Generation MOTIS Mixed-Mode Simulator," *Proceedings of the 21st Design Automation Conference*, ACM IEEE, June 25-27 1984, pp. 10-17.

[Cohen 76] Cohen, E., "Program Reference for SPICE2," ERL Memo ERL-M592, University of California, Berkeley, June 1976.

[Daniel 82] Daniel, M. E. and Gwyn, C. W., "CAD Systems for IC Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **CAD-1**(1):2-12, January 1982.

[Deutsch 84] Deutsch, J. T. and Newton, A. R., "A Multiprocessor Implementation of Relaxation Based Electrical Circuit Simulation," *Proceedings of the 21st Design Automation Conference*, ACM IEEE, June 25-27 1984, pp. 350-357.

[Doshi 84] Doshi, M. H., Sullivan, R. B. and Schuler, D. M., "THEMIS Logic Simulator - A Mix Mode, Multi-Level, Hierarchical, Interactive Digital Circuit Simulator," *Proceedings of the 21st Design Automation Conference*, ACM IEEE, June 25-27 1984, pp. 24-31.

[Dumlugol 83] Domlugol, D., de Man, H. J., Stevens, P. and Schrooten, G. G., "Local Relaxation Algorithms for Event-Driven Simulation of MOS Networks Including Assignable Delay Modeling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **CAD-2**(3):193-202, July 1983.

[Fan 77] Fan, S. P., Hsueh, M. Y., Newton, A. R. and Pederson, D. O., "MOTIS-C: A New Circuit Simulator for MOS LSI Circuits," *Proceedings of the IEEE International Symposium on Circuits and Systems*, IEEE, April 1977, pp. 700-703.

[Hafer 83] Hafer, L. J. and Parker, A. C., "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **CAD-2**(1):4-18, January 1983.

[Hill 79] Hill, D. D. and vanCleemput, W. M., "SABLE: A Tool for Generating Structured, Multi-Level Simulations," *Proceedings of the 16th Design Automation Conference*, ACM IEEE, June 25-27 1979, pp. 272-279.

[Hill 80] Hill, D. D. and vanCleemput, W. M., "SABLE: Multi-Level Simulation for Hierarchical Design," *Proceedings of the IEEE International Symposium on Circuits and Systems*, IEEE, April 1980, pp. 431-434.

[Lanthrop 85] Lanthrop, R. H. and Kirk, R. S., "An Extensible Object-Oriented Mixed-Mode Functional Simulation System," *Proceedings of the 22nd Design Automation Conference*, ACM IEEE, June 1985, pp. 630-636.

[Lewke 83] Lewke, K. and Rammig, F. J., "Description and Simulation of MOS Devices in Register Transfer Languages," Anceau, F. and Aas E. J. (Ed.), *VLSI '83*, Elsevier Science Publishers B. V., August 16-19 1983, pp. 73-83.

[Nagel 75] Nagel, L. W., "SPICE2: A Computer Program to Simulate Semiconductor Circuits," ERL Memo ERL-M520, University of California, Berkeley, May 1975.

[Nestor 82] Nestor, J. A. and Thomas, D. E., "Defining and Implementing a Multilevel Design Representation with Simulation Applications," *Proceedings of the 19th Design Automation Conference*, ACM IEEE, June 14-16 1982, pp. 740-746.

[Newton 78] Newton, A. R., "The Simulation of Large-Scale Integrated Circuits," ERL Memo ERL-M78/52, University of California, Berkeley, July 1978.

[Newton 79] Newton, A. R., "Techniques for the Simulation of Large Scale Integrated Circuits," *IEEE Transactions on Circuits and Systems* **CAS-26**(9):741-749, September 1979.

[Newton 84] Newton, A. R. and Sangiovanni-Vincentelli, A. L., "Relaxation-Based Electrical Simulation," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* **CAD-3**(4):308-331, October 1984.

[Pfister 82] Pfister, G. F., "The Yorktown Simulation Engine," *Proceedings of the 19th Design Automation Conference*, ACM IEEE, June 14-16 1982, pp. 55-59.

[Reference 85] *Reference Guide to Symbolics Lisp*, 1985.

[Solden 86] Solden, S., "Waveforms as First-Class Objects in Schema," May 1986. Bachelor of Science Thesis.

References

[Terman 83] Terman, C. J., "Simulation Tools for Digital LSI Design," Ph.D. Thesis, Massachusetts Institute of Technology, September 1983.

[Thomas 83] Thomas, D. E. and Nestor, J. A., "Defining and Implementing a Multilevel Design Representation with Simulation Applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **CAD-2**(3):135-145, July 1983.

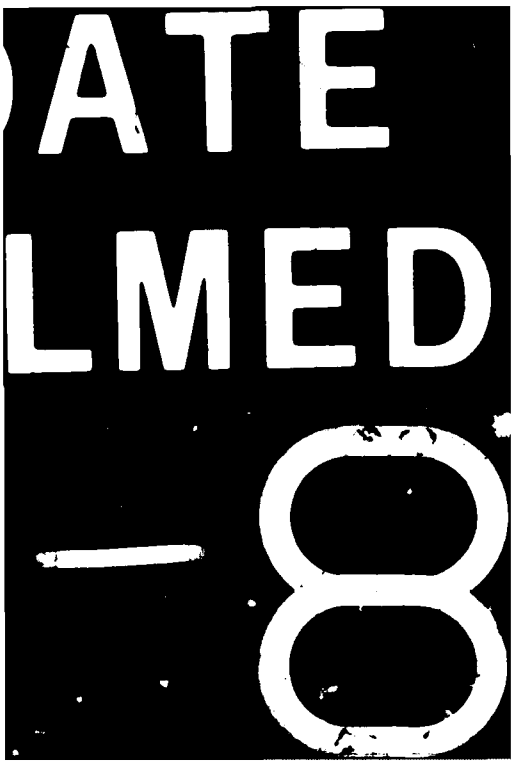
[Weeks 73] Weeks, W., et al, "Algorithms for ASTAP -- A Network Analysis Program," *IEEE Transactions on Circuit Theory* **CT-20**(6):628-634, November 1973.

[Williams 84] Williams, B. C., "Qualitative Analysis of MOS Circuits," Technical Report 767, Massachusetts Institute of Technology, July 1984.

[Zippel 85] Zippel, R. E. and Clark, G. C., "Schema - An Architecture for Knowledge Based CAD," *International Conference on Computer-Aided Design*, IEEE, November 1985, pp. 50-52.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 Copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 Copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 Copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 Copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 Copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 Copy
Dr. G. Hopper, USNR NAVDAC-OOH Department of the Navy Washington, DC 20374	1 Copy


ATE
LMED
8